# METHOD FOR MAPPING ENVIRONMENTAL RESOURCES TO MEMORY FOR PROGRAM ACCESS

## Related Applications

This application is related to U.S. Application No. *08/942,402*, entitled

5   "DIAGNOSTIC AND MANAGING DISTRIBUTED PROCESSOR SYSTEM",

Attorney Docket No. MNFRAME.005A1; U.S. Application No. *08/942,448*, entitled

"METHOD FOR MANAGING A DISTRIBUTED PROCESSOR SYSTEM", Attorney

Docket No. MNFRAME.005A2; and U.S. Application No. *08/942,222* entitled

"SYSTEM FOR MAPPING ENVIRONMENTAL RESOURCES TO MEMORY FOR

10   PROGRAM ACCESS", Attorney Docket No. MNFRAME.005A3, which are being

filed concurrently herewith.

## Priority Claim

The benefit under 35 U.S.C. § 119(e) of the following U.S. provisional

application(s) is hereby claimed:

15

| Title | Application No. | Filing Date |
|---|---|---|
| "Remote Access and Control of Environmental Management System" | 60/046,397 | May 13, 1997 |
| "Hardware and Software Architecture for Inter-Connecting an Environmental Management System with a Remote Interface" | 60/047,016 | May 13, 1997 |
| "Self Management Protocol for a Fly-By-Wire Service Processor" | 60/046,416 | May 13, 1997 |
| "Computer System Hardware Infrastructure for Hot Plugging Single and Multi-Function PC Cards Without Embedded Bridges" | 60/046,398 | May 13, 1997 |
| "Computer System Hardware Infrastructure for Hot Plugging Multi-Function PCI Cards With Embedded Bridges" | 60/046,312 | May 13, 1997 |

## Appendices

30   Appendix A, which forms a part of this disclosure, is a list of commonly

owned copending U.S. patent applications. Each one of the applications listed in

Appendix A is hereby incorporated herein in its entirety by reference thereto.

## Copyright Rights

## Background of the Invention

### Field of the Invention

The invention relates to the field of fault tolerant computer systems. More particularly, the invention relates to a managing and diagnostic system for evaluating and controlling the environmental conditions of a fault tolerant computer system.

### Description of the Related Technology

As enterprise-class servers become more powerful and more capable, they are also becoming ever more sophisticated and complex. For many companies, these changes lead to concerns over server reliability and manageability, particularly in light of the increasingly critical role of server-based applications. While in the past many systems administrators were comfortable with all of the various components that made up a standards-based network server, today's generation of servers can appear as an incomprehensible, unmanageable black box. Without visibility into the underlying behavior of the system, the administrator must "fly blind." Too often, the only indicators the network manager has on the relative health of a particular server is whether or not it is running.

It is well-acknowledged that there is a lack of reliability and availability of most standards-based servers. Server downtime, resulting either from hardware or software faults or from regular maintenance, continues to be a significant problem. By one estimate, the cost of downtime in mission critical environments has risen to an annual total of $4.0 billion for U.S. businesses, with the average downtime event resulting in a $140 thousand loss in the retail industry and a $450 thousand loss in the securities industry. It has been reported that companies lose as much as $250

thousand in employee productivity for every 1% of computer downtime. With emerging Internet, intranet and collaborative applications taking on more essential business roles every day, the cost of network server downtime will continue to spiral upward. Another major cost is of system downtime administrators to diagnose and fix the system. Corporations are looking for systems which do not require real time service upon a system component failure.

While hardware fault tolerance is an important element of an overall high availability architecture, it is only one piece of the puzzle. Studies show that a significant percentage of network server downtime is caused by transient faults in the I/O subsystem. Transient failures are those which make a server unusable, but which disappear when the server is restarted, leaving no information which points to a failing component. These faults may be due, for example, to the device driver, the adapter card firmware, or hardware which does not properly handle concurrent errors, and often causes servers to crash or hang. The result is hours of downtime per failure, while a system administrator discovers the failure, takes some action and manually reboots the server. In many cases, data volumes on hard disk drives become corrupt and must be repaired when the volume is mounted. A dismount-and-mount cycle may result from the lack of hot pluggability in current standards-based servers. Diagnosing intermittent errors can be a frustrating and time-consuming process. For a system to deliver consistently high availability, it should be resilient to these types of faults.

Modern fault tolerant systems have the functionality monitor the ambient temperature of a storage device enclosure and the operational status of other components such the cooling fans and power supply. However, a limitation of these server systems is that they do not contain self-managing processes to correct malfunctions. Thus, if a malfunction occurs in a typical server, the one corrective measure taken by the server is to give notification of the error causing event via a computer monitor to the system administrator. If the system error caused the system to stop running, the system administrator might never know the source of the error. Traditional systems are lacking in detail and sophistication when notifying system administrators of system malfunctions. System administrators are in need of a graphical user interface for monitoring the health of a network of servers.

-3-

Administrators need a simple point-and-click interface to evaluate the health of each server in the network. In addition, existing fault tolerant servers rely upon operating system maintained logs for error recording. These systems are not capable of maintaining information when the operating system is inoperable due to a system malfunction.

Existing systems also do not have an interface to control the changing or addition of an adapter. Since any user on a network could be using a particular device on the server, system administrators need a software application that will control the flow of communications to a device before, during, and after a hot plug operation on an adapter.

Also, in the typical fault tolerant computer system, the control logic for the diagnostic system is associated with a particular processor. Thus, if the environmental control processor malfunctioned, then all diagnostic activity on the computer would cease. In traditional systems, there is no monitoring of fans, and no means to make up cooling capacity lost when a fan fails. Some systems provide a processor located on a plug-in PCI card which can monitor some internal systems, and control turning power on and off. If this card fails, obtaining information about the system, and controlling it remotely, is no longer possible. Further, these systems are not able to affect fan speed or cooling capacity.

Therefore, a need exists for improvements in server management which will result in greater reliability and dependability of operation. Server users are in need of a management system by which the users can accurately gauge the health of their system. Users need a high availability system that should not only be resilient to faults, but should allow for maintenance, modification, and growth--without downtime. System users should be able to replace failed components, and add new functionality, such as new network interfaces, disk interface cards and storage, without impacting existing users. As system demands grow, organizations must frequently expand, or scale, their computing infrastructure, adding new processing power, memory, storage and I/O capacity. With demand for 24-hour access to critical, server-based information resources, planned system downtime for system service or expansion has become unacceptable.

-4-

## Summary of the Invention

Embodiments of the inventive monitoring and management system provides system administrators with new levels of client/server system availability and management. It gives system administrators and network managers a comprehensive

5    view into the underlying health of the server--in real time, whether on-site or off-site. In the event of a failure, the invention enables the administrator to learn why the system failed, why the system was unable to boot, and to control certain functions of the server.

A method of mapping environmental resources to memory, comprising

10    providing a computer, providing a microcontroller network, connecting the microcontroller network to a computer containing a central processor and a memory, creating an information pathway between the network of microcontrollers and specific memory addresses of the memory and executing commands on the at least one microcontroller which manage and diagnose system functions. A method of mapping

15    environmental resources to memory, comprising providing a computer, including a processor and a memory, connected to a microcontroller network, connecting a plurality of sensors to the microcontroller network and providing a network model in the computer memory, wherein the computer is capable of communicating with a selected one of the sensors by mapping a unique identifier to the microcontroller in

20    the network connected to the selected sensor.

## Brief Description of the Drawings

Figure 1 is one embodiment of a top-level block diagram showing a fault tolerant computer system of the invention, including mass storage and network connections.

25    Figure 2 is one embodiment of a block diagram showing a first embodiment of a multiple bus configuration connecting I/O adapters and a network of microcontrollers to the clustered CPUs of the fault tolerant computer system shown in Figure 1.

Figure 3 is one embodiment of a block diagram showing a second embodiment

30    of a multiple bus configuration connecting canisters containing I/O adapters and a

network of microcontrollers to the clustered CPUs of the fault tolerant system shown in Figure 1.

Figure 4 is one embodiment of a top-level block diagram illustrating the microcontroller network shown in Figures 2 and 3.

Figures 5A and 5B are detailed block diagrams showing one embodiment of the microcontroller network shown in Figure 4 illustrating the signals and values monitored by each microcontroller, and the control signals generated by the microcontrollers.

Figure 6 is one embodiment of a flowchart showing the process by which a remote user can access diagnostic and managing services of the microcontroller network shown in Figures 4, 5A and 5B.

Figure 7 is one embodiment of a block diagram showing the connection of an industry standard architecture (ISA) bus to the microcontroller network shown in Figures 4, 5A and 5B.

Figure 8 is one embodiment of a flowchart showing the master to slave communications of the microcontrollers shown in Figures 4, 5A and 5B.

Figure 9 is one embodiment of a flowchart showing the slave to master communications of the microcontrollers shown in Figures 4, 5A and 5B.

Figures 10A and 10B are flowcharts showing one process by which the System Interface, shown in Figure 4, 5A and 5B, gets commands and relays commands from the ISA bus to the network of microcontrollers.

Figures 11A and 11B are flowcharts showing one process by which a Chassis microcontroller, shown in Figures 4, 5A and 5B, manages and diagnoses the power supply to the computer system.

Figure 12 is a flowchart showing one process by which the Chassis controller, shown in Figures 4, 5A and 5B, monitors the addition and removal of a power supply from the fault tolerant computer system.

Figure 13 is a flowchart showing one process by which the Chassis controller, shown in Figures 4, 5A and 5B, monitors temperature.

Figure 14A and 14B are flowcharts showing one embodiment of the activities undertaken by CPU A controller, shown in Figures 4, 5A and 5B.

Figure 15 is a detailed flowchart showing one process by which the CPU A controller, show in Figures 4, 5A and 5B, monitors the fan speed for the system board of the computer.

Figure 16 is a flowchart showing one process by which activities of the CPU B controller, shown in Figures 4, 5A and 5B, scans for system faults.

Figure 17 is a flowchart showing one process by which activities of a Canister controller, shown in Figures 4, 5A and 5B, monitors the speed of the canister fan of the fault tolerant computer system.

Figure 18 is a flowchart showing one process by which activities of the System Recorder, shown in Figures 4, 5A and 5B, resets the NVRAM located on the backplane of the fault tolerant computer system.

### Detailed Description of the Invention

The following detailed description presents a description of certain specific embodiments of the present invention. However, the invention can be embodied in a multitude of different ways as defined and covered by the claims. In this description, reference is made to the drawings wherein like parts are designated with like numerals throughout.

Figure 1 is one embodiment of a block diagram showing a fault tolerant computer system of the present invention. Typically the computer system is one server in a network of servers and communicating with client computers. Such a configuration of computers is often referred to as a client-server architecture. A fault tolerant server is useful for mission critical applications such as the securities business where any computer down time can result in catastrophic financial consequences. A fault tolerant computer will allow for a fault to be isolated and not propagate through the system thus providing complete or minimal disruption to continuing operation. Fault tolerant systems also provide redundant components such as adapters so service can continue even when one component fails.

The system includes a fault tolerant computer system 100 connecting to external peripheral devices through high speed I/O channels 102 and 104. The peripheral devices communicate and are connected to the high speed I/O channels 102 and 104 by mass storage buses 106 and 107. In different embodiments of the

-7-

invention, the bus system 106, 107 could be Peripheral Component Interconnect (PCI), Microchannel, Industrial Standard Architecture (ISA) and Extended ISA (EISA) architectures. In one embodiment of the invention, the buses 106, 107 are PCI. Various kinds of peripheral controllers 108, 112, 116, and 128, may be connected to

5    the buses 106 and 107 including mass storage controllers, network adapters and communications adapters. Mass storage controllers attach to data storage devices such as magnetic disk, tape, optical disk, CD-ROM. These data storage devices connect to the mass storage controllers using one of a number of industry standard interconnects, such as small computer storage interface (SCSI), IDE, EIDE, SMD.

10   Peripheral controllers and I/O devices are generally off-the-shelf products. For instance, sample vendors for a magnetic disk controller 108 and magnetic disks 110 include Qlogic, and Quantum (respectively). Each magnetic disk may hold multiple Gigabytes of data.

A client server computer system typically includes one or more network

15   interface controllers (NICs) 112 and 128. The network interface controllers 112 and 128 allow digital communication between the fault tolerant computer system 100 and other computers (not shown) such as a network of servers via a connection 130. For LAN embodiments of the network adapter, the network media used may be, for example, Ethernet (IEEE 802.3), Token Ring (IEEE 802.5), Fiber Distributed Datalink

20   Interface (FDDI) or Asynchronous Transfer Mode (ATM).

In the computer system 100, the high speed I/O channels, buses and controllers (102-128) may, for instance, be provided in pairs. In this example, if one of these should fail, another independent channel, bus or controller is available for use until the failed one is repaired.

25   In one embodiment of the invention, a remote computer 130 is connected to the fault tolerant computer system 100. The remote computer 130 provides some control over the fault tolerant computer system 100, such as requesting system status.

Figure 2 shows one embodiment of the bus structure of the fault tolerant computer system 100. A number 'n' of central processing units (CPUs) 200 are

30   connected through a host bus 202 to a memory controller 204, which allows for access to semiconductor memory by the other system components. In one embodiment of

the invention, there are four CPUs 200, each being an Intel Pentium® Pro microprocessor. A number of bridges 206, 208 and 210 connect the host bus to three additional bus systems 212, 214, and 216. These bridges correspond to high speed I/O channels 102 and 104 shown in Figure 1. The buses 212, 214 and 216 correspond to the buses 106 and 107 shown in Figure 1. The bus systems 212, 214 and 216, referred to as PC buses, may be any standards-based bus system such as PCI, ISA, EISA and Microchannel. In one embodiment of the invention, the bus systems 212, 214, 216 are PCI. In another embodiment of the invention a proprietary bus is used.

An ISA Bridge 218 is connected to the bus system 212 to support legacy devices such as a keyboard, one or more floppy disk drives and a mouse. A network of microcontrollers 225 is also interfaced to the ISA bus 226 to monitor and diagnose the environmental health of the fault tolerant system. Further discussion of the network will be provided below.

A bridge 230 and a bridge 232 connects PC buses 214 and 216 with PC buses 234 and 236 to provide expansion slots for peripheral devices or adapters. Separating the devices 238 and 240 on PC buses 234 and 236 reduces the potential that a device or other transient I/O error will bring the entire system down or stop the system administrator from communicating with the system.

Figure 3 shows an alternative bus structure embodiment of the fault tolerant computer system 100. The two PC buses 214 and 216 contain bridges 242, 244, 246 and 248 to PC bus systems 250, 252, 254, and 256. As with the PC buses 214 and 216, the PC buses 250, 252, 254 and 256 can be designed according to any type of bus architecture including PCI, ISA, EISA, and Microchannel. The PC buses 250, 252, 254, and 256 are connected, respectively, to a canister 258, 260, 262 and 264. The canisters 258, 260, 262, and 264 are casings for a detachable bus system and provide multiple slots for adapters. In the illustrated canister, there are four adapter slots.

Referring now to Figure 4, the present invention for monitoring and diagnosing environmental conditions may be implemented by using a network of microcontrollers 225 located on the fault tolerant computer system 100. In one embodiment some of

-9-

the microcontrollers are placed on a system board or motherboard 302 while other microcontrollers are placed on a backplane 304. Furthermore, in the embodiment of Figure 3, some of the microcontrollers such as Canister controller A 324 may reside on a removable canister.

Figure 4 illustrates that the network of microcontrollers 225 is connected to one of the CPUs 200 by an ISA bus 308. The ISA 308 bus interfaces the network of microcontrollers 225 which are connected on the microcontroller bus 310 through a System Interface 312. In one embodiment of the invention, the microcontrollers communicate through an I$^2$C serial bus, also referred to as a microcontroller bus 310. The document "The I$^2$C Bus and How to Use It" (Philips Semiconductor, 1992) is hereby incorporated by reference. The I$^2$C bus is a bi-directional two-wire bus and operates at a 400 kbps rate in the present embodiment. However, other bus structures and protocols could be employed in connection with this invention. In other embodiments, IEEE 1394 (Firewire), IEEE 422, IEEE 488 (GPIB), RS-185, Apple ADB, Universal Serial Bus (USB), or Controller Area Network (CAN) could be utilized as the microcontroller bus. Control on the microcontroller bus is distributed. Each microcontroller can be a sender (a master) or a receiver (a slave) and each is interconnected by this bus. A microcontroller directly controls its own resources, and indirectly controls resources of other microcontrollers on the bus.

Here are some of the features of the I$^2$C-bus:

• Only two bus line are required: a serial data line (SDA) and a serial clock line (SCL).

• Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.

• The bus is a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.

• Serial, 8-bit oriented, bi-directional data transfers can be made at up to 400 kbit/second in the fast mode.

-10-

Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the I²C bus. Each device is recognized by a unique address and can operate as either a transmitter or receiver, depending on the function of the device. Further, each device can operate from time to time as both a transmitter

5   and a receiver. For example, a memory device connected to the I² C bus could both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers (see Table 1). A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a

10   slave.

TABLE 1  Definition of I²C-bus terminology       *Official*

| Term | Description |
|---|---|
| Transmitter | The device which sends the data to the bus |
| Receiver | The device which receives the data from the bus - |
| Master | The device which initiates a transfer, generates clock signals and terminates a transfer |
| Slave | The device addressed by a master |
| Multi-master | More than one master can attempt to control the bus at the same time without corrupting the message. Each device at separate times may act as a master. |
| Arbitration | Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted |
| Synchronization | Procedure to synchronize the clock signal of two or more devices |

15

20

25

The I²C-bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. As masters are usually

30   microcontrollers, consider the case of a data transfer between two microcontrollers

-11-

3

connected to the I²C-bus. This highlights the master-slave and receiver-transmitter relationships to be found on the I²C-bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data between microcontrollers is further described in Figure 8.

5          The possibility of connecting more than one microcontroller to the I²C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the conflict that might ensue from such an event, an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I²C interfaces to the I²C-bus.

10         If two or more masters try to put information onto the bus, as long as they put the same information onto the bus, there is no problem. Each monitors the state of the SDL. If a microcontroller expects to find that the SDL is high, but finds that it is low, the microcontroller assumes it lost the arbitration and stops sending data. The clock signals during arbitration are a synchronized combination of the clocks
15         generated by the masters using the wired-AND connection to the SCL line.

Generation of clock signal on the I²C-bus is always the responsibility of master devices. Each master microcontroller generates its own clock signals when transferring data on the bus.

In one embodiment, the command, diagnostic, monitoring and history functions
20         of the microcontroller network 102 are accessed using a global network memory and a protocol has been defined so that applications can access system resources without intimate knowledge of the underlying network of microcontrollers. That is, any function may be queried simply by generating a network "read" request targeted at the function's known global network address. In the same fashion, a function may be
25         exercised simply by "writing" to its global network address. Any microcontroller may initiate read/write activity by sending a message on the I²C bus to the microcontroller responsible for the function (which can be determined from the known global address of the function). The network memory model includes typing information as part of the memory addressing information.

30         Referring to Figure 4, in one embodiment of the invention, the network of microcontrollers 310 includes ten processors. One of the purposes of the

-12-

/3

the microcontrollers are placed on a system board or motherboard 302 while other microcontrollers are placed on a backplane 304. Furthermore, in the embodiment of Figure 3, some of the microcontrollers such as Canister controller A 324 may reside on a removable canister.

Figure 4 illustrates that the network of microcontrollers 225 is connected to one of the CPUs 200 by an ISA bus 308. The ISA 308 bus interfaces the network of microcontrollers 225 which are connected on the microcontroller bus 310 through a System Interface 312. In one embodiment of the invention, the microcontrollers communicate through an $I^2C$ serial bus, also referred to as a microcontroller bus 310. The document "The $I^2C$ Bus and How to Use It" (Philips Semiconductor, 1992) is hereby incorporated by reference. The $I^2C$ bus is a bi-directional two-wire bus and operates at a 400 kbps rate in the present embodiment. However, other bus structures and protocols could be employed in connection with this invention. In other embodiments, IEEE 1394 (Firewire), IEEE 422, IEEE 488 (GPIB), RS-185, Apple ADB, Universal Serial Bus (USB), or Controller Area Network (CAN) could be utilized as the microcontroller bus. Control on the microcontroller bus is distributed. Each microcontroller can be a sender (a master) or a receiver (a slave) and each is interconnected by this bus. A microcontroller directly controls its own resources, and indirectly controls resources of other microcontrollers on the bus.

Here are some of the features of the $I^2C$-bus:

- Only two bus line are required: a serial data line (SDA) and a serial clock line (SCL).

- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.

- The bus is a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.

- Serial, 8-bit oriented, bi-directional data transfers can be made at up to 400 kbit/second in the fast mode.

Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the $I^2C$ bus. Each device is recognized by a unique address and can operate as either a transmitter or receiver, depending on the function of the device. Further, each device can operate from time to time as both a transmitter

5 and a receiver. For example, a memory device connected to the $I^2$ C bus could both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers (see Table 1). A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a

10 slave.

TABLE 1  Definition of $I^2C$-bus terminology

| Term | Description |
| --- | --- |
| Transmitter | The device which sends the data to the bus |
| Receiver | The device which receives the data from the bus |
| Master | The device which initiates a transfer, generates clock signals and terminates a transfer |
| Slave | The device addressed by a master |
| Multi-master | More than one master can attempt to control the bus at the same time without corrupting the message. Each device at separate times may act as a master. |
| Arbitration | Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted |
| Synchronization | Procedure to synchronize the clock signal of two or more devices |

The $I^2C$-bus is a multi-master bus. This means that more than one device

30 capable of controlling the bus can be connected to it. As masters are usually microcontrollers, consider the case of a data transfer between two microcontrollers

connected to the I²C-bus. This highlights the master-slave and receiver-transmitter relationships to be found on the I²C-bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data between microcontrollers is further described in Figure 8.

5        The possibility of connecting more than one microcontroller to the I²C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the conflict that might ensue from such an event, an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I²C interfaces to the I²C-bus.

10       If two or more masters try to put information onto the bus, as long as they put the same information onto the bus, there is no problem. Each monitors the state of the SDL. If a microcontroller expects to find that the SDL is high, but finds that it is low, the microcontroller assumes it lost the arbitration and stops sending data. The clock signals during arbitration are a synchronized combination of the clocks

15       generated by the masters using the wired-AND connection to the SCL line.

Generation of clock signal on the I²C-bus is always the responsibility of master devices. Each master microcontroller generates its own clock signals when transferring data on the bus.

In one embodiment, the command, diagnostic, monitoring and history functions

20       of the microcontroller network 102 are accessed using a global network memory and a protocol has been defined so that applications can access system resources without intimate knowledge of the underlying network of microcontrollers. That is, any function may be queried simply by generating a network "read" request targeted at the function's known global network address. In the same fashion, a function may be

25       exercised simply by "writing" to its global network address. Any microcontroller may initiate read/write activity by sending a message on the I²C bus to the microcontroller responsible for the function (which can be determined from the known global address of the function). The network memory model includes typing information as part of the memory addressing information.

30       Referring to Figure 4, in one embodiment of the invention, the network of microcontrollers 310 includes ten processors. One of the purposes of the

microcontroller network 225 is to transfer messages to the other components of the server system 100. The processors or microcontrollers include: a System Interface 312, a CPU A controller 314, a CPU B controller 316, a System Recorder 320, a Chassis controller 318, a Canister A controller 324, a Canister B controller 326, a Canister C controller 328, a Canister D controller 330 and a Remote Interface controller 332. The System Interface controller 312, the CPU A controller 314 and the CPU B controller 316 are located on a system board 302 in the fault tolerant computer system 100. Also located on the system board are one or more central processing units (CPUs) or microprocessors 164 200 and the Industry Standard Architecture (ISA) bus 296 226 that connects to the System Interface Controller 312. The CPUs 200 may be any conventional general purpose single-chip or multi-chip microprocessor such as a Pentium7, Pentium® Pro or Pentium® II processor available from Intel Corporation, A MIPS® processor available from Silicon Graphics, Inc., a SPARC processor from Sun Microsystems, Inc., a Power PC® processor available from Motorola, or an ALPHA® processor available from Digital Equipment Corporation. In addition, the CPUs 200 may be any conventional special purpose microprocessor such as a digital signal processor or a graphics processor.

The System Recorder 320 and Chassis controller 318, along with a data string such as a random access non-volatile access memory (NVRAM) 322 that connects to the System Recorder 320, are located on a backplane 304 of the fault tolerant computer system 100. The data storage 322 may be independently powered and may retain its contents when power is unavailable. The data storage 322 is used to log system status, so that when a failure of the computer 100 occurs, maintenance personnel can access the storage 322 and search for information about what component failed. An NVRAM is used for the data storage 322 in one embodiment but other embodiments may use other types and sizes of storage devices.

The System Recorder 320 and Chassis controller 318 are the first microcontrollers to power up when server power is applied. The System Recorder 320, the Chassis controller 318 and the Remote Interface microcontroller 332 are the three microcontrollers that have an independent bias 5 Volt power supplied to them if main server power is off. This independent bias 5 Volt power is provided by a

Remote Interface Board (not shown).  The Canister controllers 324-330 are not considered to be part of the backplane 304 because  each is mounted on a card attached to the canister.

5 　　　Figures 5A and 5B are one embodiment of a block diagram that illustrates some of the signal lines that are used by the different microcontrollers.  Some of the signal lines connect to actuators and other signal lines connect to sensors.  In one embodiment of the invention the microcontrollers in the network are commercially available microcontrollers.   Examples of off-the-shelf microcontrollers are the PIC16c65 and the PIC16c74  available from Microchip Technology Inc, the 8051
10 from Intel Corporation, the 8751 available from Atmel, and a P80CL580 microprocessor available from Philips, could be utilized.

　　　The Chassis controller 318 is connected to a set of temperature detectors 502, 504, and 506 which read the temperature on the backplane 304 and the system board 302.  Figure 5 also illustrates the signal lines that connect the System Recorder 320
15 to the NVRAM 322 and a timer chip 520.  In one embodiment of the invention, the System Recorder 320 is the only microcontroller that can access the NVRAM 322. The Canister controller 324 is connected to a Fan Tachometer Signal Mux 508 which is used to detect the speed of the fans.  The CPU A controller 314 also is connected to a fan mux 508 which gathers the fan speed of system fans.  The CPU A controller
20 314 displays errors to a user by writing to an LCD display 512.  Any microcontroller can request the CPU A controller 314 to write a message to the LCD display 512. The System Interface 312 is connected to a response buffer 514 which queues outgoing response signals in the order that they are received.  Similarly, a request signal buffer 516 is connected to the System Interface 312 and stores, or queues
25 request signals in the order that they are received.

　　　Software applications can access the network of microcontrollers 225 by using the software program header file that is listed at the end of the specification in the section titled "Header File for Global Memory Addresses."  This header file provides a global memory address for each function of the microcontroller network 225.  By
30 using the definitions provided by this header file, applications can request and send

-14-

information to the microcontroller network 225 without needing to know where a particular sensor or activator resides in the microcontroller network.

Figure 6 is one embodiment of a flowchart illustrating the process by which under one implementation of the present invention, a remote application connected, say, through the connection of Figure 1, can access the network of microcontrollers 225. Starting at state 600, a remote software application, such as a generic system management application like Hewlett-Packard Open View, or an application specific to this computer system, retrieves a management information block (MIB) object by reading and interpreting a MIB file, or by an application's implicit knowledge of the MIB object's structure. This retrieval could be the result of an operator using a graphical user interface (GUI), or as the result of some automatic system management process. The MIB is a description of objects, which have a standard structure, and contain information specific to the MIB object ID associated with a particular MIB object. At a block 602, the remote application builds a request for information by creating a request which references a particular MIB object by its object ID, sends the request to the target computer using a protocol called SNMP (simple network management protocol). SNMP is a type of TCP/IP protocol. Moving to state 604, the remote software sends the SNMP packet to a local agent Microsoft WinSNMP, for example, which is running on the fault tolerant computer system 100, which includes the network of microcontrollers 225 (Figure 4). The agent is a specialized program which can interpret MIB object IDs and objects. The local agent software runs on one of the CPUs 200 of Figures 2 and 3.

The local agent examines the SNMP request packet (state 606). If the local agent does not recognize the request, the local agent passes the SNMP packet to an extension SNMP agent. Proceeding to state 608, the extension SNMP agent dissects the object ID. The extension SNMP agent is coded to recognize from the object ID, which memory mapped resources managed by the network of microcontrollers need to be accessed (state 608). The agent then builds the required requests for the memory mapped information in the command protocol format understood by the network of microcontrollers 225. The agent then forwards the request to a microcontroller network device driver (state 610).

-15-

The device driver then sends the information to the network of microcontrollers 225 at state 612. The network of microcontrollers 225 provides a result to the device driver in state 614. The result is returned to the extension agent, which uses the information to build the MIB object, and return it to the ~~extension~~ SNMP agent (state 616). The local SNMP agent forwards the MIB object via SNMP to the remote agent (state 616). Finally, in state 620, the remote agent forwards the result to the remote application software.

For example, if a remote application needs to know the speed of a fan, the remote application reads a file to find the object ID for fan speed. The object ID for the fan speed request may be "837.2.3.6.2". Each set of numbers in the object ID represent hierarchical groups of data. For example the number "3" of the object ID represents the cooling system. The "3.6" portion of the object ID represents the fans in the cooling. All three numbers "3.6.2" indicate speed for a particular fan in a particular cooling group.

In this example, the remote application creates a SNMP packet containing the object ID to get the fan speed on the computer 100. The remote application then sends the SNMP packet to the local agent. Since the local agent does not recognize the fan speed object ID, the local agent forwards the SNMP packet to the extension agent. The extension agent parses the object ID to identify which specific memory mapped resources of the network of microcontrollers 225 are needed to build the MIB object whose object ID was just parsed. The extension agent then creates a message in the command protocol required by the network of microcontrollers 225. A device driver which knows how to communicate requests to the network of microcontrollers 225 takes this message and relays the command to the network of microcontrollers 225. Once the network of microcontrollers 225 finds the fan speed, it relays the results to the device driver. The device driver passes the information to the extension agent. The agent takes the information supplied by the microcontroller network device driver and creates a new SNMP packet. The local agent forwards this packet to the remote agent, which then relays the fan speed which is contained in the packet to the remote application program.

Figure 7 is one embodiment of a block diagram of the interface between the network of microcontrollers 225 and the ISA bus 308 of Figures 2 and 3. The interface to the network of microcontrollers 225 includes a System Interface processor 312 which receives event and request signals, processes these signals, and transmits command, status and response signals to the operating system of the CPUs 200. In one embodiment, the System Interface processor 312 is a PIC16C65 controller chip, available from Microchip, Technology Inc., which includes an event memory (not shown) organized as a bit vector, having at least sixteen bits. Each bit in the bit vector represents a particular type of event. Writing an event to the System Interface processor 312 sets a bit in the bit vector that represents the event. Upon receiving an event signal from another microcontroller, the System Interface 312 interrupts CPUs 200. Upon receiving the interrupt, the CPUs 200 will check the status of the System Interface 312 to ascertain that an event is pending. Alternatively, the CPUs 200 may periodically poll the status of the System Interface 312 to ascertain whether an event is pending. The CPUs 200 may then read the bit vector in the System Interface 312 to ascertain the type of event that occurred and thereafter notify a system operator of the event by displaying an event message on a monitor connected to the fault tolerant computer 100 or another computer in the server network. After the system operator has been notified of the event, as described above, she may then obtain further information about the system failure which generated the event signal by accessing the NVRAM 322.

The System Interface 312 communicates with the CPUs 200 by receiving request signals from the CPUs 200 and sending response signals back to the CPUs 200. Furthermore, the System Interface 312 can send and receive status and command signals to and from the CPUs 200. For example, a request signal may be sent from a software application inquiring as to whether the System Interface 312 has received any event signals, or inquiring as to the status of a particular processor, subsystem, operating parameter. The following discussion explains how in further detail at the state 612, the device driver sends the request to the network of microcontrollers, and then, how the network of microcontrollers returns the result (state 614). A request signal buffer 516 is connected to the System Interface 312 and stores, or queues,

-17-

request signals in the order that they are received, first in-first out (FIFO). Similarly, a response buffer 514 is connected to the System Interface 312 and queues outgoing response signals in the order that they are received (FIFO). These queues are one byte wide, (messages on the I²C bus are sequences of 8-bit bytes, transmitted bit serially

5      on the SDL).

A message data register (MDR) 707 is connected to the request and response buffers 516 and 514 and controls the arbitration of messages to and from the System Interface 312 via the request and response buffers 516 and 514. In one embodiment, the MDR 707 is eight bits wide and has a fixed address which may be accessed by

10      the server's operating system via the ISA bus 226 connected to the MDR 707. As shown in Figure 7, the MDR 707 has an I/O address of 0CC0h. When software application running on one of the CPUs 200 desires to send a request signal to the System Interface 312, it does so by writing a message one byte at a time to the MDR 707. The application then indicates to the system interface processor 312 that the

15      command has been completely written, and may be processed.

The system interface processor 312 writes the response one byte at a time to the response queue, then indicates to the CPU (via an interrupt or a bit in the status register) that the response is complete, and ready to be read. The CPU 200 then reads the response queue one byte at a time by reading the MDR 707 until all bytes of the

20      response are read.

The following is one embodiment of the command protocol used to communicate with the network of microcontrollers 225.

**Table 2 Command Protocol Format**

| READ REQUEST FORMAT | | | WRITE REQUEST FORMAT | | |
|---|---|---|---|---|---|
| Offset | | | Offset | | |
| Byte 0 | Slave Addr (7 bits) | 0 LSBit | Byte 0 | Slave Addr (7 bits) | 0 LSBit |
| Byte 1 | MSBit (1) | Type | Byte 1 | MSBit (0) | Type |
| Byte 2 | Command ID (LSB) | | Byte 2 | Command ID (LSB) | |

-18-

| | |
|---|---|
| Byte 3 | Command ID (MSB) |
| Byte 4 | Read Request Length (N) |
| Byte 5 | Check Sum |

| | |
|---|---|
| Byte 3 | Command ID (MSB) |
| Byte 4 | Write Request Length (N) |
| Byte 5 | Data Byte 1 |
| : | : |
| Byte N+4 | Data Byte N |
| Byte N+5 | Check Sum |

**READ RESPONSE FORMAT**

Offset

| | | |
|---|---|---|
| Byte 0 | Slave Addr (7 bits) | 1 LSBit |
| Byte 1 | Read Response Length (N) | |
| Byte 2 | Data Byte 1 | |
| : | : | |
| Byte N+1 | Data Byte N | |
| Byte N+2 | Status | |
| Byte N+3 | Check Sum | |
| Byte N+4 | Inverted Slave Addr | |

**WRITE RESPONSE FORMAT**

Offset

| | | |
|---|---|---|
| Byte 0 | Slave Addr (7 bits) | 1 LSBit |
| Byte 1 | Write Response Length (0) | |
| Byte 2 | Status | |
| Byte 3 | Check Sum | |
| Byte 4 | Inverted Slave Addr | |

The following is a description of each of the fields in the command protocol.

**Table 3**

**Description of Command Protocol Fields**

| FIELD | DESCRIPTION |
|---|---|

| | | |
|---|---|---|
| | Slave Addr | Specifies the processor identification code. This field is 7 bits wide. Bit [7...1]. |
| | LSBit | Specifies what type of activity is taking place. If LSBit is clear (0), the master is writing to a slave. If LSBit is set (1), the master is reading from a slave. |
| 5 | | |
| | MSBit | Specifies the type of command. It is bit 7 of byte 1 of a request. If this bit is clear (0), this is a write command. If it is set (1), this is a read command. |
| | Type | Specifies the data type of this command, such as bit or string. |
| 10 | | |
| | Command ID (LSB) | Specifies the least significant byte of the address of the processor. |
| | Command ID (MSB) | Specifies the most significant byte of the address of the processor. |
| 15 | Length (N) | |
| | Read Request | Specifies the length of the data that the master expects to get back from a read response. The length, which is in bytes, does not include the Status, Check Sum, and Inverted Slave Addr fields. |
| 20 | Read Response | Specifies the length of the data immediately following this byte, that is byte 2 through byte N+1. The length, which is in bytes, does not include the Status, Check Sum, and Inverted Slave Addr fields. |
| | Write Request | Specifies the length of the data immediately following this byte, that is byte 2 through byte N+1. The length, which is in bytes, does not include the Status, Check Sum, and Inverted Slave Addr fields. |
| 25 | | |
| | Write Response | Always specified as 0. |
| | Data Byte 1 | Specifies the data in a read request and response, and a write request. |
| 30 | | |
| | Data Byte N | |

2 1

| | |
|---|---|
| Status | Specifies whether or not this command executes successfully. A non-zero entry indicates a failure. |
| Check Sum | Specifies a direction control byte to ensure the integrity of a message on the wire. |
| Inverted Slave Addr | Specifies the Slave Addr, which is inverted. |

The System Interface 312 further includes a command and status register (CSR) 709 which initiates operations and reports on status. The operation and functionality of CSR 709 is described in further detail below. Both synchronous and asynchronous I/O modes are provided by the System Interface 312. During a synchronous mode of operation, the device driver waits for a request to be completed. During an asynchronous mode of operation the device driver sends the request, and asks to be interrupted when the request completes. To support asynchronous operations, an interrupt line 711 is connected between the System Interface 312 and the ISA bus 226 and provides the ability to request an interrupt when asynchronous I/O is complete, or when an event occurs while the interrupt is enabled. As shown in Figure 7, in one embodiment, the address of the interrupt line 711 is fixed and indicated as IRQ 15 which is an interrupt address number used specifically for the ISA bus 226.

The MDR 707 and the request and response buffers 516 and 514, respectively, transfer messages between a software application running on the CPUs 200 and the failure reporting system of the invention. The buffers 516 and 514 have two functions: (1) they store data in situations where one bus is running faster than the other, i.e., the different clock rates, between the ISA bus 226 and the microcontroller bus 310; and (2) they serve as interim buffers for the transfer of messages -- this relieves the System Interface 312 of having to provide this buffer.

When the MDR 707 is written to by the ISA bus 226, it loads a byte into the request buffer 226. When the MDR 707 is read from the ISA bus 516, it unloads a byte from the response buffer 514. The System Interface 312 reads and executes messages from buffer 516 when a message command is received in the CSR 709. A response message is written to the response buffer 514 when the System Interface 312 completes executing the command. The system operator receives a completed message

over the microcontroller bus 310. A software application can read and write message data to and from the buffers 516 and 514 by executing read and write instructions through the MDR 707.

The CSR 709 has two functions. The first is to initiate commands, and the second is to report status. The System Interface commands are usually executed synchronously. That is, after issuing a command, the microcontroller network device driver should continue to poll the CSR 709 status to confirm command completion. In addition to synchronous I/O mode, the microcontroller network device driver can also request an asynchronous I/O mode for each command by setting a "Asyn Req" bit in the command. In this mode, an interrupt is generated and sent to the ISA bus 226, via the interrupt line 711, after the command has completed executing.

In the described embodiment, the interrupt is asserted through IRQ15 of the ISA programmable interrupt controller (PIC). The ISA PIC interrupts the CPU 200s when a signal transitioning from high to low, or from low to high, is detected at the proper input pin (edge triggered). Alternatively, the interrupt line 711 may utilize connect to a level-triggered input. A level-triggered interrupt request is recognized by keeping the signal at the same level, or changing the level of a signal, to send an interrupt. The microcontroller network device driver can either enable or disable interrupts by sending "Enable Ints" and "Disable Ints" commands to the CSR 701. If the interrupt 711 line is enabled, the System Interface 312 asserts the interrupt signal IRQ15 of the PIC to the ISA bus 226, either when an asynchronous I/O is complete or when an event has been detected.

In the embodiment shown in Figure 2, the System Interface 312 may be a single-threaded interface. Since messages are first stored in the queue, then retrieved from the queue by the other side of the interface, a device driver should write one message, containing a sequence of bytes, at a time. Thus, only one message should be in progress at a time using the System Interface 312. Therefore, a program or application must allocate the System Interface 312 for its use before using it, and then de-allocate the interface 514 when its operation is complete. The CSR 709 indicates which operator is allocated access to the System Interface 312.

Referring to Figures 2 and 7, an example of how messages are communicated between the System Interface 312 and CPUs 200 in one embodiment of the invention is as follows (all byte values are provided in hexadecimal numbering). A system management program (not shown) sends a command to the network of microcontrollers 225 to check temperature and fan speed. To read the temperature from CPU A controller 314 the program builds a message for the device driver to forward to the network of microcontrollers 225. First, the device driver on CPUs 200 allocates the interface by writing the byte "01" to the CSR 709. If another request was received, the requestor would have to wait until the previous request was completed. To read the temperature from Chassis controller 318 the device driver would write into the request queue 516 through the MDR 707 the bytes "02 83 03 00 FF". The first byte "02" would signify to the System Interface 312 that a command is intended for the Chassis controller 318. The first bits of the second byte "83" indicates that a master is writing to a slave. The last or least significant three bits of the byte "83" indicate the data type of the request. The third and fourth bytes "03 00" indicate that the read request temperature function of the Chassis controller 318 is being requested. The final byte "FF" is the checksum.

After writing the bytes to the MDR 707, a "13" (message command) is written by the device driver to the CSR 709, indicating the command is ready to be executed. The System Interface processor 312 passes the message bytes to the microcontroller bus 310, receives a response, and puts the bytes into the response FIFO 514. Since there is only one system interface processor 312, there is no chance that message bytes will get intermingled.

After all bytes are written to the response FIFO, the System Interface processor 312 sets a bit in the CSR 709 indicating message completion. If directed to do so by the device driver, the system interface 312 asserts an interrupt on IRQ 15 upon completion of the task.

The CPUs 200 would then read from the response buffer 516 through the MDR 707 the bytes "02 05 27 3C 27 26 27 00". The first byte in the string is the slave address shown as Byte 0 in the Read Response Format. The first byte 02 indicates that the CPU A Chassis controller 318 was the originator of the message. The second

-23-

byte "05" indicates the number of temperature readings that follow. The second Byte "05" maps to Byte 1 of the Read Response Format. In this example, the Chassis controller 318 returned five temperatures. The second reading, byte "3C" (60 decimal) is above normal operational values. The last byte "00" is a check sum which

5    is used to ensure the integrity of a message.

The CPUs 200 agent and device driver requests the fan speed by writing the bytes "03 83 04 00 FF" to the network of microcontroller 225. Each byte follows the read request format specified in Table 2. The first byte "03" indicates that the command is for the CPU A Controller 314. The second byte "83" indicates that the

10    command is a read request of a string data type.

A response of "03 06 41 43 41 42 41 40 00" would be read from MDR 707 by the device driver. The first byte "03" indicates to the device driver that the command is from the CPU A controller 314. The speed bytes "41 43 41 42 41 40" indicate the revolutions per second of a fan in hexadecimal. The last byte read from

15    the MDR 707 "00" is the checksum.

Since one of the temperatures is higher than the warning threshold, 55°C, and fan speed is within normal (low) range, a system administrator or system management software may set the fan speed to high with the command bytes "03 01 01 00 01 01 FF". The command byte "03" indicates that the command is for the CPU A 314. The

20    first byte indicates that a write command is requested. The third and fourth bytes, which correspond to byte 2 and 3 of the write request format, indicate a request to increase the fan speed. The fifth byte, which corresponds to byte 4 of the write request format indicates to the System Interface 312 that one byte is being sent. The sixth byte contains the data that is being sent. The last byte "FF" is the checksum.

25    Figure 8 is one embodiment of a flowchart describing the process by which a master microcontroller communicates with a slave microcontroller. Messages between microcontrollers can be initiated by any microcontroller on the microcontroller bus 310 (Figure 4). A master microcontroller starts out in state 800.

In state 802, the microcontroller arbitrates for the start bit. If a microcontroller

30    sees a start bit on the microcontroller bus 310, it cannot gain control of the microcontroller bus 310. The master microcontroller proceeds to state 804. In the

-24-

state 804, the microcontroller increments a counter every millisecond. The microcontroller then returns to state 800 to arbitrate again for the start bit. If at state 806 the count reaches 50 ms, the master has failed to gain the bus (states 808 and 810). The microcontroller then returns to the state 800 to retry the arbitration process.

If in the state 802, no start bit is seen on the microcontroller bus 310, the microcontroller bus 310 is assumed to be free (i.e., the microcontroller has successfully arbitrated won arbitration for the microcontroller bus 310). The microcontroller sends a byte at a time on the microcontroller bus 310 (state 812). After the microcontroller has sent each byte, the microcontroller queries the microcontroller bus 310 to insure that the microcontroller bus 310 is still functional. If the SDA and SCL lines of the microcontroller bus 310 are not low, the microcontroller is sure that the microcontroller bus 310 is functional and proceeds to state 816. If the SDA and SCL lines are not drawn high, then the microcontroller starts to poll the microcontroller bus 310 to see if it is functional. Moving to state 819, the microcontroller increments a counter Y and waits every 22 microseconds. If the counter Y is less than five milliseconds (state 820), the state 814 is reentered and the microcontroller bus 310 is checked again. If the SDA and SCL lines are low for 5 milliseconds (indicated when, at state 820, the counter Y exceeds 5 milliseconds), the microcontroller enters state 822 and assumes there is a microcontroller bus error. The microcontroller then terminates its control of the microcontroller bus 310 (state 824).

If in the state 814, the SDA/SCL lines do not stay low (state 816), the master microcontroller waits for a response from a slave microcontroller (state 816). If the master microcontroller has not received a response, the microcontroller enters state 826. The microcontroller starts a counter which is incremented every one millisecond. Moving to state 828, if the counter reaches fifty milliseconds, the microcontroller enters state 830 indicating a microcontroller bus error. The microcontroller then resets the microcontroller bus 310 (state 832).

Returning to state 816, if the master microcontroller does receive a response in state 816, the microcontroller enters state 818 and receives the data from the slave

26

microcontroller. At state 820, the master microcontroller is finished communicating with the slave microcontroller.

Figure 9 is one embodiment of a block diagram illustrating the process by which a slave microcontroller communicates with a master microcontroller. Starting in state 900, the slave microcontroller receives a byte from a master microcontroller. The first byte of an incoming message always contains the slave address. This slave address is checked by all of the microcontrollers on the microcontroller bus 310. Whichever microcontroller matches the slave address to its own address handles the request.

At a decision state 902, an interrupt is generated on the slave microcontroller. The microcontroller checks if the byte received is the first received from the master microcontroller (state 904). If the current byte received is the first byte received, the slave microcontroller sets a bus time-out flag (state 906). Otherwise, the slave microcontroller proceeds to check if the message is complete (state 908). If the message is incomplete, the microcontroller proceeds to the state 900 to receive the remainder of bytes from the master microcontroller. If at state 908, the slave microcontroller determines that the complete message has been received, the microcontroller proceeds to state 909.

Once the microcontroller has received the first byte, the microcontroller will continue to check if there is an interrupt on the microcontroller bus 310. If no interrupt is posted on the microcontroller bus 310, the slave microcontroller will check to see if the bus time-out flag is set. The bus time-out flag is set once a byte has been received from a master microcontroller. If in the decision state 910 the microcontroller determines that the bus time-out flag is set, the slave microcontroller will proceed to check for an interrupt every 10 milliseconds up to 500 milliseconds. For this purpose, the slave microcontroller increments the counter every 10 milliseconds (state 912). In state 914, the microcontroller checks to see if the microcontroller bus 310 has timed out. If the slave microcontroller has not received additional bytes from the master microcontroller, the slave microcontroller assumes that the microcontroller bus 310 is hung and resets the microcontroller bus 310 (state

916). Next, the slave microcontroller aborts the request and awaits further requests from other master microcontrollers (state 918).

Referring to the state 909, the bus timeout bit is cleared, and the request is processed and the response is formulated. Moving to state 920, the response is sent a byte at a time. At state 922, the same bus check is made as was described for the state 814. States 922, 923 and 928 form the same bus check and timeout as states 814, 819 and 820. If in state 928 this check times out, a bus error exists, and this transaction is aborted (states 930 and 932).

Figures 10A and 10B are flow diagrams showing one process by which the System Interface 312 handles requests from other microcontrollers in the microcontroller network and the ISA bus 226 (Figures 4 and 5). The System Interface 312 relays messages from the ISA bus 226 to other microcontrollers in the network of microcontrollers 225. The System Interface 312 also relays messages from the network of microcontrollers to the ISA bus 226.

Referring to Figures 10A and 10B, the System Interface 312 initializes all variables and the stack pointer (state 1000). Moving to state 1002, the System Interface 312 starts its main loop in which it performs various functions. The System Interface 312 next checks the bus timeout bit to see if the microcontroller bus 310 has timed-out (decision state 1004). If the microcontroller bus 310 has timed-out, the System Interface 312 resets the microcontroller bus 310 in state 1006.

Proceeding to a decision state 1008, the System Interface 312 checks to see if any event messages have been received. An event occurs when the System Interface 312 receives information from another microcontroller regarding a change to the state of the system. At state 1010, the System Interface 312 sets the event bit in the CSR 709 to one. The System Interface 312 also sends an interrupt to the operating system if the CSR 709 has requested interrupt notification.

Proceeding to a decision state 1012, the System Interface 312 checks to see if a device driver for the operating system has input a command to the CSR. If the System Interface 312 does not find a command, the System Interface 312 returns to state 1002. If the System Interface does find a command from the operating system, the System Interface parses the command. For the "allocate command", the System

Interface 312 resets the queue to the ISA bus 226 resets the done bit in the CSR 709 (state 1016) and sets the CSR Interface Owner ID (state 1016). The Owner ID bits identify which device driver owns control of the System Interface 312.

For the "de-allocate command", the System Interface 312 resets the queue to the ISA bus 226, resets the done bit in the CSR 709, and clears the Owner ID bits (state 1018).

For the "clear done bit command" the System Interface 312 clears the done bit in the CSR 709 (state 1020). For the "enable interrupt command" the System Interface 312 sets the interrupt enable bit in the CSR 709 (state 1022). For the "disable interrupt command," the System Interface 312 sets the interrupt enable bit in the CSR 709 (state 1024). For the "clear interrupt request command", the System Interface 312 clears the interrupt enable bit in the CSR 709 (state 1026).

If the request from the operating system was not meant for the System Interface 312, the command is intended for another microcontroller in the network 225. The only valid command remaining is the "message command." Proceeding to state 1028, the System Interface 312 reads message bytes from the request buffer 516. From the state 1028, the System Interface 312 proceeds to a decision state 1030 in which the System Interface 312 checks whether the command was for itself. If the command was for the System Interface 312, moving to state 1032, the System Interface 312 processes the command. If the ID did not match an internal command address, the System Interface 312 relays the command the appropriate microcontroller (state 1034) by sending the message bytes out over the microcontroller bus 310.

Figures 11A and 11B are flowcharts showing an embodiment of the functions performed by the Chassis controller 318. Starting in the state 1100, the Chassis controller 318 initializes its variables and stack pointer.

Proceeding to state 1102, the Chassis controller 318 reads the serial numbers of the microcontrollers contained on the system board 302 and the backplane 304. The Chassis controller 318 also reads the serial numbers for the Canister controllers 324, 326, 328 and 330. The Chassis controller 318 stores all of these serial numbers in the NVRAM 322.

Next, the Chassis controller 318 start its main loop in which it performs various diagnostics (state 1104). The Chassis controller 318 checks to see if the microcontroller bus 310 has timed-out (state 1106). If the bus has timed-out, the Chassis controller 318 resets the microcontroller bus 310 (state 1008). If the microcontroller bus 310 has not timed out the Chassis controller proceeds to a decision state 1110 in which the Chassis controller 318 checks to see if a user has pressed a power switch.

If the Chassis controller 318 determines a user has pressed a power switch, the Chassis controller changes the state of the power to either on or off (state 1112). Additionally, the Chassis controller logs the new power state into the NVRAM 322.

The Chassis controller 318 proceeds to handle any power requests from the Remote Interface 332 (state 1114). As shown in Figure 9, a power request message to this microcontroller is received when the arriving message interrupts the microcontroller. The message is processed and a bit is set indicating request has been made to toggle power. At state 1114, the Chassis controller 318 checks this bit. If the bit is set, the Chassis controller 318 toggles the system, i.e., off-to-on or on-to-off, power and logs a message into the NVRAM 322 that the system power has changed state (state 1116).

Proceeding to state 1118, the Chassis controller 318 checks the operating system watch dog counter for a time out. If the Chassis controller 318 finds that the operating system has failed to update the timer, the Chassis controller 318 proceeds to log a message with the NVRAM 322 (state 1120). Additionally, the Chassis controller 318 sends an event to the System Interface 312 and the Remote Interface 332.

Since it takes some time for the power supplies to settle and produce stable DC power, the Chassis controller delays before proceeding to check DC (state 1122).

The Chassis controller 318 then checks for changes in the canisters 258-264 (state 1124), such as a canister being inserted or removed. If a change is detected, the Chassis controller 318 logs a message to the NVRAM 322 (state 1126). Additionally, the Chassis controller 318 sends an event to the System Interface 312 and the Remote Interface 332.

30

The Chassis controller 318 proceeds to check the power supply for a change in status (state 1128). The process by which the Chassis controller 318 checks the power supply is described in further detail in the discussion for Figure 12.

The Chassis controller then checks the temperature of the system (state 1132). The process by which the Chassis controller 318 checks the temperature is described in further detail in the discussion for Figure 13.

At state 1136, the Chassis controller 318 reads all of the voltage level signals. The Chassis controller 318 saves these voltage levels values in an internal register for reference by other microcontrollers.

Next, the Chassis controller 318 checks the power supply signals for AC/DC changes (state 1138). If the Chassis controller 318 detects a change in the Chassis controller 318, the Chassis controller 318 logs a message to the NVRAM 322 (state 1140). Additionally, the Chassis controller 318 sends an event to the System Interface 312 and the Remote Interface 332 that a AC/DC signal has changed. The Chassis controller 318 then returns to state 1104 to repeat the monitoring process.

Figure 12 is a flowchart showing one process by which the Chassis controller 318 checks the state of the redundant power supplies termed number 1 and 2. These power supplies are monitored and controlled by the chassis controller 318 through the signal lines shown in Figure 5A. When a power supply fails or requires maintenance, the other supply maintains power to the computer 100. To determine whether a power supply is operating properly or not, its status of inserted or removed (by maintenance personnel) should be ascertained. Furthermore, a change in status should be recorded in the NVRAM 322. Figure 12 describes in greater detail the state 1128 shown in Figure 11B.

Starting in state 1202, the Chassis controller 318 checks the power supply bit. If the power supply bit indicates that a power supply should be present, the Chassis controller checks whether power supply "number 1" has been removed (state 1204). If power supply number 1 has been removed, the chassis microcontroller 318 checks whether its internal state indicates power supply number one should be present. If the internal state was determined to be present, then the slot is checked to see whether power supply number 1 is still physically present (state 1204). If power supply

number 1 has been removed, the PS_PRESENT#1 bit is changed to not present (state 1208). The Chassis controller 318 then logs a message in the NVRAM 322.

Referring to state 1206, if the PS_PRESENT#1 bit indicates that power supply number 1 is not present, the Chassis controller 318 checks whether power supply number 1 has been inserted (i.e., checks to see if it is now physically present) (state 1206). If it has been inserted, the Chassis controller 318 then logs a message into the NVRAM 322 that the power supply number 1 has been inserted (state 1210) and changes the value of PS_PRESENT#1 to present.

After completion, states 1204, 1206, 1208, and 1210 proceed to state 1212 to monitor power supply number 2. The Chassis controller 318 checks whether the PS_PRESENT#2 bit is set to present. If the PS_PRESENT#2 bit indicates that power supply "number 2" should be there, the Chassis controller 318 proceeds to state 1224. Otherwise, the Chassis controller 318 proceeds to state 1226. At state 1224, the Chassis controller 318 checks if power supply number 2 is still present. If power supply number 2 has been removed, the Chassis controller 318 logs in the NVRAM 322 that power supply number 2 has been removed (state 1228). The chassis controller also changes the value of PS_PRESENT#2 bit to not present.

Referring to decision state 1226, if the PS_PRESENT#2 bit indicates that no power supply number 2 is present, the Chassis controller 318 checks if power supply number 2 has been inserted. If so, the Chassis controller 318 then logs a message into the NVRAM 322 that power supply number 2 has been inserted and changes the value of PS_PRESENT#2 to present (state 1230). After completion of states 1224, 1226, 1228, and 1230, the chassis controller 318 proceeds to state 1232 to monitor the AC/DC power supply changed signal.

If in decision state 1234 the Chassis controller 318 finds that the AC/DC power supply changed signal from the power supplies is asserted, the change in status is recorded in state 1236. The Chassis controller 318 continues the monitoring process by proceeding to the state 1132 in Figure 11B.

Figure 13 is a flowchart showing one process by which the Chassis controller 318 monitors the temperature of the system. As shown in Figure 5A, the Chassis controller 318 receives temperature detector signal lines from five temperature

-31-

3 2

detectors located on the backplane and the motherboard. If either component indicates it is overheating, preventative action may be taken manually, by a technician, or automatically by the network of microcontrollers 225. Figure 13 describes in greater detail the state 1132 shown in Figure 11B.

To read the temperature of the Chassis, the Chassis controller 318 reads the temperature detectors 502, 504, and 506 (state 1300). In the embodiment of the invention shown in Figure 13 there are five temperature detectors (two temperature detectors not shown). Another embodiment includes three temperature detectors as shown.

The Chassis controller 318 checks the temperature detector 502 to see if the temperature is less than -25°C or if the temperature is greater than or equal to 55°C (state 1308). Temperatures in this range are considered normal operating temperatures. Of course, other embodiments may use other temperature ranges. If the temperature is operating inside normal operating boundaries, the Chassis controller 318 proceeds to state 1310. If the temperature is outside normal operating boundaries, the Chassis controller 318 proceeds to state 1312. At state 1312, the Chassis controller 318 evaluates the temperature a second time to check if the temperature is greater than or equal to 70°C or less than or equal to -25°C. If the temperature falls below or above outside of these threshold values, the Chassis controller proceeds to state 1316. Temperatures in this range are considered so far out of normal operating temperatures, that the computer 100 should be shutdown. Of course, other temperature ranges may be used in other embodiments.

Referring to state 1316, if the temperature level reading is critical, the Chassis controller 318 logs a message in the NVRAM 322 that the system was shut down due to excessive temperature. The Chassis controller 318 then proceeds to turn off power to the system in state 1320, but may continue to operate from a bias or power supply.

Otherwise, if the temperature is outside normal operating temperatures, but only slightly deviant, the Chassis controller 318 sets a bit in the temperature warning status register (state 1314). Additionally, the Chassis controller 318 logs a message in the NVRAM 322 that the temperature is reaching dangerous levels (state 1318).

33

The Chassis controller 318 follows the aforementioned process for each temperature detector on the system. Referring back to state 1310, which was entered after determining a normal temperature from one of the temperature detectors, the Chassis controller 318 checks a looping variable "N" to see if all the sensors were read. If all sensors were not read, the Chassis controller 318 returns to state 1300 to read another temperature detector. Otherwise, if all temperature detectors were read, the Chassis controller 318 proceeds to state 1322. At state 1322, the Chassis controller 318 checks a warning status register (not shown). If no bit is set in the temperature warning status register, the Chassis controller 318 returns to the state 1136 in Figure 11B. If the Chassis controller 318 determines that a bit in the warning status register was set for one of the sensors, the Chassis controller 318 proceeds to recheck all of the sensors (state 1324). If the temperature of the sensors are still at a dangerous level, the Chassis Controller 318 maintains the warning bits in the warning status register. The Chassis controller 318 then proceeds to the state 1136 (Figure 11B). At state 1324, if the temperatures of the sensors are now at normal operating values, the Chassis controller 318 proceeds to clear all of the bits in the warning status register (state 1326). After clearing the register, the Chassis controller 318 proceeds to state 1328 to log a message in the NVRAM 322 that the temperature has returned to normal operational values, and the Chassis controller 318 proceeds to the state 11136 (Figure 11B).

Figures 14A and 14B are flowcharts showing the functions performed by one embodiment of the CPU A controller 314. The CPU A controller 314 is located on the system board 302 and conducts diagnostic checks for: a microcontroller bus timeout, a manual system board reset, a low system fan speed, a software reset command, general faults, a request to write to flash memory, checks system flag status, and a system fault.

The CPU A controller 314, starting in state 1400, initializes its variables and stack pointer. Next, in state 1402 the CPU A controller 314 starts its main loop in which it performs various diagnostics which are described below. At state 1404, the CPU A controller 314 checks the microcontroller bus 310 for a time out. If the microcontroller bus 310 has timed out, the CPU A controller 314 resets the

34

microcontroller bus 310 (state 1406). From either state 1404 or 1406, the CPU A controller 314 proceeds to check whether the manual reset switch (not shown) is pressed on the system board 302 (decision state 1408). If the CPU A controller 314 determines that the manual reset switch is pressed, the CPU A controller resets system board by asserting a reset signal (state 1410).

From either state 1408 or 1410, the CPU A controller 314 proceeds to check the fan speed (decision state 1412). If any of a number of fans speed is low (see Figure 15 and discussion below), the CPU A controller 314 logs a message to NVRAM 322 (state 1414). Additionally, the CPU A controller 314 sends an event to the Remote Interface 334 and the System Interface 312. The CPU A controller 314 next proceeds to check whether a software reset command was issued by either the computer 100 or the remote computer 132 (state 1416). If such a command was sent, the CPU A controller 314 logs a message in NVRAM 322 that system software requested the reset command (state 1418). Additionally, the CPU A controller 314 also resets the system bus 202.

From either state 1416 or 1418, the CPU A controller 314 checks the flags bits (not shown) to determine if a user defined system fault occurred (state 1420). If the CPU A controller 314 determines that a user defined system fault occurred, the CPU A controller 314 proceeds to display the fault on an LCD display 512 (Figure 5B) (state 1422).

From either state 1420 or 1422 the CPU A controller 314 proceeds to a state 1424 (if flash bit was not enabled) to check the flash enable bit maintained in memory on the CPU B controller 316. If the flash enable bit is set, the CPU A controller 314 displays a code for flash enabled on the LCD display 512. The purpose of the flash enable bit is further described in the description for the CPU B controller 316 (Figure 16).

From either state 1424 or 1426 (if the flash bit was not enabled), the CPU A controller 314 proceeds to state 1428 and checks for system faults. If the CPU A controller 314 determines that a fault occurred, the CPU A controller 314 displays the fault on the LCD display 512 (state 1430). From state 1428 if no fault occurred, or from state 1430, the CPU A controller 314 proceeds to the checks the system status

flag located in the CPU A controller's memory (decision state 1432). If the status flag indicates an error, the CPU A controller 314 proceeds to state 1434 and displays error information on the LCD display 512.

From either state 1432 or 1434, the CPU controller proceeds to state 1402 to repeat the monitoring process.

Figure 15 is a flowchart showing one process by which the CPU A controller 314 monitors the fan speed. Figure 15 is a more detailed description of the function of state 1412 in Figure 14A. Starting in state 1502, the CPU A controller 314 reads the speed of each of the fans 1506, 1508, and 1510. The fan speed is processed by a Fan Tachometer Signal Mux 508 (also shown in Figure 5B) which updates the CPU A controller 314. The CPU A controller 314 then checks to see if a fan speed is above a specified threshold (state 1512). If the fan speed is above the threshold, the CPU A controller 314 proceeds to state 1514. Otherwise, if the fan speed is operating below a specified low speed limit, the CPU A controller 314 proceeds to state 1522.

On the other hand, when the fan is operating above the low speed limit at state 1514, the CPU A controller 314 checks the hot_swap_fan register (not shown) if the particular fan was hot swapped. If the fan was hot swapped, the CPU A controller 314 proceeds to clear the fan's bit in both the fan_fault register (not shown) and the hot_swap_fan register (state 1516). After clearing these bits, the CPU A controller 314 checks the fan fault register (state 1518). If the fan fault register is all clear, the CPU A controller 314 proceeds to set the fan to low speed (state 1520) and logs a message to the NVRAM 322. The CPU A controller 314 then proceeds to state 1536 to check for a temperature warning.

Now, referring back to state 1522, if a fan speed is below a specified threshold limit, the CPU A controller 314 checks to see if the fan's speed is zero. If the fan's speed is zero, the CPU A controller 314 sets the bit in the hot_swap_fan register in state 1524 to indicate that the fan has a fault and should be replaced. If the fan's speed is not zero, the CPU A controller 314 will proceed to set a bit in the fan_fault register (state 1526). Moving to state 1528, the speed of any fans still operating is increased to high, and a message is written to the NVRAM 322.

-35-

36

In one alternative embodiment, the system self-manages temperature as follows: from either state 1520 or 1528, the CPU A controller 314 moves to state 1536 and checks whether a message was received from the Chassis controller 318 indicating temperature warning. If a temperature warning is indicated, and if there are no fan faults involving fans in the cooling group associated with the warning, the speed of fans in that cooling group is increased to provide more cooling capacity (state 1538).

Proceeding to state 1530 from either state 1536 or 1538, the CPU A controller 314 increments a fan counter stored inside of microcontroller memory. If at state 1531, there are more fans to check, the CPU A controller 314 returns to state 1502 to monitor the speed of the other fans. Otherwise, the CPU controller 314 returns to state 1416 (Figure 14).

Figure 16 is one embodiment of a flow diagram showing the functions performed by the CPU B controller 316. The CPU B controller 316 scans for system faults, scans the microcontroller bus 310, and provides flash enable. The CPU B controller 316, starting at state 1600, initializes its variables and stack pointer.

After initializing its internal state, the CPU B controller 316 enters a diagnostic loop at state 1602. The CPU B controller 316 then checks the microcontroller bus 310 for a time out (decision state 1604). If the microcontroller bus 310 has timed out, the CPU B controller 316 resets the microcontroller bus 310 in state 1606. If the microcontroller bus 310 has not timed out (state 1604) or after state 1606, the CPU B controller 316 proceeds to check the system fault register (not shown) (decision state 1608).

If the CPU B controller 316 finds a system fault, the CPU B controller 316 proceeds to log a message into the NVRAM 322 stating that a system fault occurred (state 1610). The CPU B controller 316 then sends an event to the System Interface 312 and the Remote Interface 332. Additionally, the CPU B controller 316 turns on one of a number of LED indicators 518 (Figure 5B).

If no system fault occurred, or from state 1610, the CPU B controller 316 scans the microcontroller bus 310 (decision state 1612). If the microcontroller bus 310 is hung then the CPU B controller 316 proceeds to flash an LED display 512 that the

-36-

37

microcontroller bus 310 is hung (state 1614). Otherwise, if the bus is not hung the CPU B controller 316 then proceeds to state 1624.

The CPU B controller 316 proceeds to check for a bus stop bit time out (decision state 1624). If the stop bit has timed out, the CPU B controller 316 generates a stop bit on the microcontroller bus for error recovery in case the stop bit is inadvertently being held low by another microcontroller (state 1626).

From either state 1624 or 1626, the CPU B controller 316 proceeds to check the flash enable bit to determine if the flash enable bit (not shown) is set (state 1628). If the CPU B controller 316 determines that the flash enable bit is set (by previously having received a message requesting it), the CPU B controller 316 proceeds to log a message to the NVRAM 322 (state 1630). A flash update is performed by the BIOS if the system boot disk includes code to update a flash memory (not shown). The BIOS writes new code into the flash memory only if the flash memory is enabled for writing. A software application running on the CPUs 200 can send messages requesting that BIOS flash be enabled. At state 1630, the 12 Volts needed to write the flash memory is turned on or left turned on. If the flash enable bit is not on, control passes to state 1629, where the 12 Volts is turned off, disabling writing of the flash memory.

From either state 1629 or 1630, the CPU B controller 316 proceeds to repeat the aforementioned process of monitoring for system faults (state 1602).

Figure 17 is one embodiment of a flowchart showing the functions performed by the Canister controllers 324, 326, 328 and 330 shown in Figures 4 and 5. The Canister controllers 324, 326, 328 and 330 examine canister fan speeds, control power to the canister, and determine which canister slots contain cards. The Canister controllers 324-330, starting in state 1700, initialize their variables and stack pointers.

Next, in state 1702 the Canister controllers 324-330 start their main loop in which they performs various diagnostics, which are further described below. The Canister controllers 324-330 check the microcontroller bus 310 for a time out (state 1704). If the microcontroller bus 310 has timed out, the Canister controllers 324-330 reset the microcontroller bus 310 in state 1706. After the Canister controller 324-330 reset the microcontroller bus 310, or if the microcontroller bus 310 has not timed out,

the Canister controllers 324-330 proceed to examine the speed of the fans (decision state 1708). As determined by tachometer signal lines connected through a fan multiplexer 508 (Figure 5), if either of two canister fans is below the lower threshold, the event is logged, an event is sent to the System Interface 312 and, speed, in a self-management embodiment, the fan speed is set to high. The Canister controllers 324-330 check the fan speed again, and if they are still low the canister controlling 324-330 signal a fan fault and register an error message in the NVRAM 322 (state 1710).

If the Canister controller received a request message to turn on or off canister power, a bit would have been previously set. If the Canister controllers 324-330 find this bit set (state 1712), they turn the power to the canister on, and light the canister's LED. If the bit is cleared, power to the canister is turned off, as is the LED (state 1714).

Next, the Canister controllers 324-330 read a signal for each slot which indicates whether the slot contains an adapter (state 1716). The Canister controllers 324-330 then returns to the state 1702, to repeat the aforementioned monitoring process.

Figure 18 is one embodiment of a flowchart showing the functions performed by the System Recorder controller 320. The System Recorder controller 320 maintains a system log in the NVRAM 322. The System Recorder 320 starting in state 1800 initializes its variables and stack pointer.

Next, at state 1802 the System Recorder 320 starts its main loop in which the System Recorder 320 performs various functions, which are further described below. First, the System Recorder 320 checks the microcontroller bus 310 for a time out (state 1804). If the microcontroller bus 310 has timed out, the System Recorder 320 resets the microcontroller bus 310 in state 1806. After the System Recorder 320 resets the bus, or if the microcontroller bus 310 has not timed out, the System Recorder 320 checks to see if another microcontroller had requested the System Recorder 320 to reset the NVRAM 322 (state 1808). If requested, the System Recorder 320 proceeds to reset all the memory in the NVRAM 322 to zero (decision state 1810). After resetting the NVRAM 322, or if no microcontroller had requested such a reset, the

System Recorder 320 proceeds to a get the real time clock every second from a timer chip 520 (Figure 5A) (decision state 1812).

From time to time, the System Recorder 320 will be interrupted by the receipt of messages. When these messages are for storing data in the NVRAM 322, they are carried out as they are received and the messages are stored in the NVRAM 322. Thus, there is no state in the flow of Figure 18 to explicitly store messages. The System Recorder then returns to the state 1802 to repeat the aforementioned monitoring process.

While the above detailed description has shown, described, and pointed out the fundamental novel features of the invention as applied to various embodiments, it will be understood that various omissions and substitutions and changes in the form and details of the system illustrated by be made by those skilled in the art, without departing from the intent of the invention.

+T,0410

```
       #ifndef SDL_TYPES
       #ifndef FAR_POINTERS
5      typedef unsigned char *BYTEADDRESS;
       typedef unsigned short *WORDADDRESS;
       typedef unsigned long *LONGADDRESS;
       typedef char *SBYTEADDRESS;
       typedef short *SWORDADDRESS;
10     typedef long *SLONGADDRESS;
       #else
       typedef unsigned long BYTEADDRESS;
       typedef unsigned long WORDADDRESS;
       typedef unsigned long LONGADDRESS;
15     typedef unsigned long SBYTEADDRESS;
       typedef unsigned long SWORDADDRESS;
       typedef unsigned. long SLONGADDRESS;
       #endif
       #define SDL_TYPES 1
20     #endif

       /* */
       /* $Module CS9000WS.SDL$ */
       /* */
25     /* Copyright 1996 */
       /* By NetFRAME Systems Inc. */
       /*    Milpitas, California U.S.A. */
       /* */
       /* $Author:   Ken Nguyen  $ */
30     /* $Date:   31 Mar 1997 15:28:08  $ */
       /* $Revision */
       /* */
       /* $Description$ */
       /* This file contains the NetFRAME Wire Service message and interface definition. */
35     /*  for the CS9000 */
       /* $EndDescription$ */
       /* */
       /*   Revision History */
       /* $Log:   P:/inc/cs9000ws.sdl   $ */
40     /*    */
       /*     Rev 1.16   31 Mar 1997 15:28:08   Ken Nguyen */
       /*   Added WSEvent variables, Severity bytes and WS commands. */
       /*    */
       /*     Rev 1.15   28 Jan 1997 16:31:32   Ken Nguyen */
45     /*   Cleaned up SDL file */
       /*  Added Buffer Event Commands and Event ID Number. */
       /*    */
       /*     Rev 1.14   27 Nov 1996 14:10:12   Ken Nguyen */
       /*   Added commands for Raptor 8 */
50     /*   Added WSEVENT_CPU event. */
       /*    */
       /*     Rev 1.13   25 Oct 1996 16:48:18   Ken Nguyen */
       /*   Fixed a Problem of Canister Fan Fault Status. */
       /*    */
```

41

```
/*      Rev 1.10   10 Oct 1996 16:33:04   Ken Nguyen */
/*   Added a command to count Log entry. */
/*    */
/*      Rev 1.9   30 Sep 1996 18:42:50   Ken Nguyen */
/*   Added Canister Fault Commands */
/*    */
/*      Rev 1.8   30 Sep 1996 17:34:16   Karl Johnson */
/*   Added definitions for remote interface serial protocol */
/*   Added NVRAM error counter */
/*    */
/*      Rev 1.7   13 Sep 1996 11:22:22   Ken Nguyen */
/*   Corrected Temperature data length */
/*    */
/*      Rev 1.6   09 Sep 1996 17:24:48   Karl Johnson */
/*   Added WS_SYSLOG_CLOCK  - the clock used by the log recorder to time stamp */
/*    */
/*      Rev 1.5   20 Aug 1996 01:08:36   Karl Johnson */
/*   Added screen event and corrected BOOTDEVS name. */
/*    */
/*      Rev 1.4   01 Aug 1996 15:32:50   Karl Johnson */
/*   Cleanup and added new status values. */
/*    */
/*      Rev 1.3   26 Jul 1996 17:14:38   Karl Johnson */
/*   Reduced maximum number of event types. */
/*   Added a Success Status. */
/*    */
/*      Rev 1.2   08 Jul 1996 15:57:32   Karl Johnson */
/*   Changed read write bit in datatype definition. */
/*   Added WS_BOOTDEVS missed in translating specification. */
/*    */
/*      Rev 1.1   19 Jun 1996 14:15:28   Karl Johnson */
/*   Added LCD low level access items. */
/*    */
/*      Rev 1.0   18 Jun 1996 14:06:58   Karl Johnson */
/*   Initial revision. */
/*    */
/* *********************************************** */
/*  This is the Wire Service Message format */

#ifndef PIC_PROCESSOR


struct WSMessage
   {
   unsigned char ToProcesor;
   unsigned char Type_RW;
   unsigned char AddressLow;
   unsigned char AddressHi;
   unsigned char WriteLength;
/*    WriteData   BLOCK_BYTE  0   ; Write data stream goes here */
   };
#define WSMessage_S 5

struct WSResponse
   {
```

-41-

```
            unsigned char FromProcesor;
            unsigned char ReadLength;
            /*   ReadData      BLOCK_BYTE 0   ; Read data stream goes here */
            unsigned char Status;
            };
        #define WSResponse_S  3


        #endif


        /*  */
        /*  Wire Service Local Interface Definitions */
        /*  */
        /*  Command (CSR Write) Register definitions */
        #define WSCMD_RequestInt       0x80   /*  Request interrupt on command complete */


        #define WSCMD_Allocate1        0x01   /*  Allocate interface as ID 1 */
        #define WSCMD_Allocate2        0x02   /*  Allocate interface as ID 2 */
        #define WSCMD_Allocate3        0x03   /*  Allocate interface as ID 3 */
        #define WSCMD_Allocate4        0x04   /*  Allocate interface as ID 4 */
        #define WSCMD_Allocate5        0x05   /*  Allocate interface as ID 5 */
        #define WSCMD_Allocate6        0x06   /*  Allocate interface as ID 6 */
        #define WSCMD_Allocate7        0x07   /*  Allocate interface as ID 7 */
        #define WSCMD_Deallocate       0x10   /*  Deallocate interface */
        #define WSCMD_EnableInts       0x11   /*  Enable interrupts for events */
        #define WSCMD_DisableInts      0x12   /*  Disable interrupts for events */
        #define WSCMD_Message          0x13   /*  Process message in FIFO and set done */
        #define WSCMD_ClearDone        0x20   /*  Clear done bit & error bit and clear FIFOs */
        #define WSCMD_ClearIntReq      0x21   /*  Clear Interrupt Request bit */
        /*  ( Must poll WSTS_IntReq => 0 for completion )   */
        #define WSCMD_Reset    0x0a5  /*  Reset interface */
        #define WSCMD_DiagMode         0x05a  /*  Enter Diagnostic mode */
        #define WSCMD_ExitDiagMode     0x00   /*  Exit Diagnostic mode */


        /*  Status (CSR Read) Register definitions */
        #define WSSTS_Error      0x80   /*  Error processing command */
        #define WSSTS_IntEna     0x40   /*  Event Interrupts are enabled */
        #define WSSTS_Events     0x20   /*  One or more events occured */
        #define WSSTS_Done       0x10   /*  Message command is done */
        #define WSSTS_IntReq     0x08   /*  Interrupt is being requested */
        #define WSSTS_AllocMask          0x07   /*  ID of owner of interface */


        /*  IO Addresses of Wire Service Local Interface */
        #define WSLOC_Data       0x0CC0
        #define WSLOC_CSR        0x0CC1


        /*  **************************************************** */
        /*  These are the data type definitions */

        #define WSTYPE_BIT       0x01
        #define WSTYPE_BYTE      0x02
        #define WSTYPE_STRING            0x03
        #define WSTYPE_LOG       0x04
        #define WSTYPE_EVENT             0x05
        #define WSTYPE_QUEUE             0x06
        #define WSTYPE_ARRAY             0x07
```

-42-

```c
#define WSTYPE_LOCK    0x08
#define WSTYPE_SCREEN          0x09


#define WSOP_READ      0x80
#define WSOP_WRITE     0x00



#define WSEVENT_CAN_CHG        0x01
#define WSEVENT_PS_CHG         0x02
#define WSEVENT_QUEUE          0x03
#define WSEVENT_TEMP           0x04
#define WSEVENT_ACOK           0x05
#define WSEVENT_DCOK           0x06
#define WSEVENT_FAN   0x07
#define WSEVENT_SCREEN         0x08
#define WSEVENT_CPU   0x09
#define WSEVENT_OS_TimeOut     0x0A    /* Event of OS's Timer is timed out */
#define WSEVENT_PCI_TimeOut            0x0B    /* Event of Power ON/OFF PCI Slot is timed out */
#define WSEVENT_CALLOUT        0x0C    /* Call Out Event */
#define WSEVENT_MAXVALUE              0x0F    /* Make sure no event values exceed this value */

#define WSERR_NONE     0x00    /* No error occurred */
#define WSERR_NONODE           0x01    /* Slave addressed did not respond */
#define WSERR_NOADDRESS        0x02    /* Slave responded that it had no such type/address */
#define WSERR_CORRUPTED        0x03    /* Message or Response is not valid */
#define WSERR_UNDERRUN         0x04    /* Message could not be completely transmitted or recieved
*/
#define WSERR_DATACHECK        0x05    /* Message data checksum recieved incorrectly ( try again if
possible ) */
#define WSERR_OPERATION        0x06    /* Slave operation not possible (e.g. Wr to R/O) */
#define WSERR_NODATA          0x07    /* Slave responded no data available at address (queue/log)
*/


#define WSPID_SYSLOG           0x01
#define WSPID_BACKPLANE        0x02
#define WSPID_SYSTEMA          0x03
#define WSPID_SYSTEMB          0x04
#define WSPID_LOCAL_IF         0x10
#define WSPID_REMOTE_IF        0x11
#define WSPID_CANISTER1        0x20
#define WSPID_CANISTER2        0x21
#define WSPID_CANISTER3        0x22
#define WSPID_CANISTER4        0x23


/* ******************************************************** */
/*  Wire Service Remote Interface Protocol Constants */

#define WSRI_SOM       0x7B    /*  Serial Start Of Message */
#define WSRI_EOM       0x7D    /*  Serial End Of Message */
#define WSRI_SUB       0x5C    /*  Serial Substitute next character */
#define WSRI_EVT       0x5E    /*  Serial Event indicator */

#define WSRI_REQ_IDENTIFY      0x01    /*  Request Identity and reset sequence */
#define WSRI_REQ_SECURE        0x02    /*  Request to enter Security mode (logon) */
#define WSRI_REQ_UNSECURE      0x03    /*  Request to leave Security mode */
```

44

```c
#define WSRI_REQ_MESSAGE        0x04    /* Request contains WS message to process */
#define WSRI_REQ_POLL           0x05    /*  Request status */

#define WSRI_STAT_OK            0x01    /* Request OK return data valid */
#define WSRI_STAT_OK_EVENT              0x02    /* Request OK return data valid ( Event(s) pending
) */
#define WSRI_STAT_E_SEQUENCE            0x03    /* Request not in Sequence */
#define WSRI_STAT_E_DATACHECK           0x03    /* Request check byte not correct */
#define WSRI_STAT_E_FORMAT              0x04    /* Request format incorrect */
#define WSRI_STAT_E_SECURE              0x05    /* Request requires Security mode */


/* ********************************************************** */
/*  Wire Service Log Message Constants */
/*  */
/*  First byte of log message data: Severity Level Byte */

#define WSLOG_LEVEL_UNKNOWN             0x00    /*  Unknown */
#define WSLOG_LEVEL_INFO        0x10    /*  Informational */
#define WSLOG_LEVEL_WARN        0x20    /*  Warning */
#define WSLOG_LEVEL_ERROR               0x30    /*  Error */
#define WSLOG_LEVEL_FATAL               0x40    /*  Severe/Fatal Error */


/*  Second byte of log message data: Source/Encoding Byte */
/*  - which entity logged the entry in the 4 high bits */
/*  - which type of encoding of the message is used in the 4 low bits of the byte. */
#define WSLOG_SRC_INTERNAL              0x00    /*  Wire Service Internal */
#define WSLOG_SRC_OBDIAG        0x10    /*  Onboard Diagnostics */
#define WSLOG_SRC_EXDIAG        0x20    /*  External Diagnostics */
#define WSLOG_SRC_BIOS          0x30    /*  BIOS */
#define WSLOG_SRC_DOS           0x40    /*  DOS */
#define WSLOG_SRC_WIN           0x50    /*  Windows,Win95 */
#define WSLOG_SRC_WINNT         0x60    /*  Windows/NT */
#define WSLOG_SRC_NETWARE               0x70    /*  NetWare */

#define WSLOG_TYPE_BINARY       0x00    /*  Message data is Binary */
#define WSLOG_TYPE_ASCII        0x10    /*  Message data is ASCII */
#define WSLOG_TYPE_UNICODE              0x20    /*  Message data is Unicode */


/* ********************************************************** */
/*  This is the Wire Service addresses for named items. */
/*  */
/*  Addresses are composed of three parts: Processor ID, Data Type and Subaddress */
/*  In this table the address is encoded as a 4 bytes in hexadecimal notation: */
/*  PPTTAAAAh where PP is the processor ID, TT is the data type and AL AH is the */
/*  2 byte subaddress. Processor ID's 00 and 20 are special, 00 applies to all */
/*  processors and 20 applies to all canister processors. */
/*  */
/*                                   PPTTALAH */
#define WS_DESCRIPTION          0x00030100      /* (S) Wire Service Processor Type/Description */
#define WS_REVISION     0x00030200      /* (S) Wire Service Software Revision/Date Info */
#define WS_WDOG_CALLOUT    0x01010200       /* (L) This is a bit controlling callout on a wathcdog
timeout. */
#define WS_WDOG_RESET           0x01010300      /* (L) This is a bit controlling system on a wathcdog
timeout. */
```

```
#define WS_NVRAM_RESET          0x01020100    /* (B) Trigger to reset NVRAM Data */
#define WS_SYS_BOOTFLAG1        0x01020200    /* (B) System Boot Flag 1 */
#define WS_SYS_BOOTFLAG2        0x01020300    /* (B) System Boot Flag 2 */
#define WS_SYS_BOOTFLAG3        0x01020400    /* (B) System Boot Flag 3 */
#define WS_SYS_BOOTFLAG4        0x01020500    /* (B) System Boot Flag 4 */
#define WS_SYS_XDATA_KBYTES     0x01020600    /* (B) Size of the WS_SYS_XDATA in
kilobytes */
#define WS_NVRAM_FAULTS         0x01020700    /* (B) Faults detected in NVRAM Data */
#define WS_SYS_XDATA            0x01070000    /* Byte Array for storage of arbitrary external data
in NVRAM */
#define WS_SYS_LOG              0x01040000    /* System Log */
#define WS_RI_QUEUE             0x01060100    /* (Q) Queue of data going to Remote Interface */
#define WS_SI_QUEUE             0x01060200    /* (Q) Queue of data going to System Interface */
#define WS_SYS_SCREEN           0x01090000    /* System Screen */
#define WS_CALLOUT_SCRIPT       0x01030300    /* (S) The callout script for remote notification */
#define WS_PASSWORD             0x01030400    /* (S) The access password for Wire Service */
#define WS_SYS_BP_SERIAL        0x01030500    /* (S) Last known Back Plane serial data */
#define WS_SYS_CAN_SERIAL1      0x01030600    /* (S) Last known Canister 1 Serial data */
#define WS_SYS_CAN_SERIAL2      0x01030700    /* (S) Last known Canister 2 Serial data */
#define WS_SYS_CAN_SERIAL3      0x01030800    /* (S) Last known Canister 3 Serial data */
#define WS_SYS_CAN_SERIAL4      0x01030900    /* (S) Last known Canister 4 Serial data */
#define WS_SYS_RI_SERIAL        0x01031600    /* (S) Last known Remote Interface serial data */
#define WS_SYS_SB_SERIAL        0x01031700    /* (S) Last known System Board serial data */
#define WS_SYS_PS_SERIAL1       0x01031800    /* (S) Last known Power Supply 1 serial data */
#define WS_SYS_PS_SERIAL2       0x01031900    /* (S) Last known Power Supply 2 serial data */
#define WS_SYS_PS_SERIAL3       0x01031a00    /* (S) Last known Power Supply 3 serial data */
#define WS_NAME                 0x01031b00    /* (S) System Identifying Name */
#define WS_BOOTDEVS             0x01031c00    /* (S) BIOS Boot drive information */
#define WS_SYS_LOG_CLOCK        0x01031d00    /* (S) Current time from log timestamp clock (seconds)
*/
#define WS_SYS_LOG_COUNT        0x01031e00    /* (S) Number of Log Entry */
#define WS_MODEM_INIT           0x01031f00    /* (S) Modem initialization string */
#define WS_EVENT_ID01           0x01032000    /* (S) Canister Change Event */
#define WS_EVENT_ID02           0x01032100    /* (S) Power Supply Change Event */
#define WS_EVENT_ID03           0x01032200    /* (S) Queue Event */
#define WS_EVENT_ID04           0x01032300    /* (S) Temp Warn or Shut Event */
#define WS_EVENT_ID05           0x01032400    /* (S) ACOK Change Event */
#define WS_EVENT_ID06           0x01032500    /* (S) DCOK Change Event */
#define WS_EVENT_ID07           0x01032600    /* (S) Fan Fault Event */
#define WS_EVENT_ID08           0x01032700    /* (S) Screen Event */
#define WS_EVENT_ID09           0x01032800    /* (S) CPU Fault Event */
#define WS_EVENT_ID0A           0x01032900    /* (S) OS_TimeOut Event */
#define WS_CALLOUT_MASK         0x01034000    /* (S) Call Out Masking string */
#define WS_BIOS_REV             0x01034100    /* (S) Storage of current BIOS Revision */
#define WS_SYS_POWER            0x02010100    /* (L) Controls system master power S4_POWER_ON
*/
#define WS_SYS_REQ_POWER        0x02010200    /* (L) Set to request main power on */
#define WS_BP_P12V              0x02020100    /* (B) Analog Measure of +12 volt main supply */
#define WS_BP_P3V               0x02020200    /* (B) Analog Measure of +3.3 volt main supply */
#define WS_BP_N12V              0x02020300    /* (B) Analog Measure of -12 volt main supply */
#define WS_BP_P5V               0x02020400    /* (B) Analog Measure of +5 volt main supply */
#define WS_BP_VREF              0x02020500    /* (B) Analog Measure of VREF */
#define WS_SYS_BP_TYPE          0x02020600    /* (B) Type of system backplane currently only two
types Type 0= 4 canister (small) and Type 1= 8 canister (large) */
```

```
#define WS_SYS_CAN_PRES        0x02020700      /* (B) Presence bits for canisters (LSB=1, MSB=8)
*/
#define WS_SYS_PS_ACOK         0x02020800      /* (B) Power supply ACOK status (LSB=1, MSB=3)
*/
#define WS_SYS_PS_DCOK         0x02020900      /* (B) Power supply DCOK status (LSB=1, MSB=3)
*/
#define WS_SYS_PS_PRES         0x02020a00      /* (B) Presence bits for power supplies (LSB=1,
MSB=3) */
#define WS_SYS_RSTIMER         0x02020b00      /* (B) Used to delay reset/run until power stabilized
*/
#define WS_SYS_TEMP_SHUT       0x02020c00      /* (B) Shutdown temperature. Initialized to ??? */
#define WS_SYS_TEMP_WARN       0x02020d00      /* (B) Warning temperature. Initialized to ??? */
#define WS_SYS_WDOG            0x02020e00      /* (B) System watchdog timer */

/* First issues following command in phase 2 */

#define WS_OS_RESOLUTION_16         0x04020600      /* (B) Set Resolution (0,1,2,3) of Timer1 */
#define WS_OS_COUNTER_16       0x04020700      /* (B) Set Counter from (00 - FFh) of Timer1 */

/* If either operation's failed that it will response error code "02h" back, then try raptor 8 and future
command */

#define WS_OS_RESOLUTION_8          0x02020f00      /* (B) Set Resolution (0,1,2,3) of Timer1 */
#define WS_OS_COUNTER_8        0x02021000      /* (B) Set Counter from (00 - FFh) of Timer1 */

/* If it's failed it is raptor 16 phase 1 that does not support watchdog */

#define WS_SYS_TEMP_DATA       0x02030300      /* (S) Temperatures of all sensors on temperature bus
in address order */
#define WS_SB_FAN_HI   0x03010100      /* (L) System Board Fans HI */
#define WS_SB_FAN_LED          0x03010200      /* (L) System Board Fan Fault LED */
#define WS_SYS_RUN     0x03010300      /* (L) Controls the system halt/run line S1_OK_TO_RUN. */
#define WS_SYS_SB_TYPE         0x03010400      /* (L) Set System Type (0: Raptor16 or 1:Raptor 8)
*/
#define WS_SB_BUSCORE          0x03020200      /* (B) System Board BUS/CORE speed ratio to use
on reset */
#define WS_SB_FANFAULT         0x03020300      /* (B) System Board Fan fault bits */
#define WS_SB_FAN_LOLIM        0x03020400      /* (B) Fan speed low speed fault limit */
#define WS_SB_LCD_COMMAND           0x03020500      /* (B) Low level LCD Controller Command
*/
#define WS_SB_LCD_DATA         0x03020600      /* (B) Low level LCD Controller Data */
#define WS_LCD_MSG     0x03020700      /* (B) Send a Byte of Fault Bits from Monitor-B to Monitor-A
*/
#define WS_SB_DIMM_TYPE        0x03030300      /* (S) The type of DIMM in each DIMM socket as
a 16 byte string */
#define WS_SB_FAN_DATA         0x03030400      /* (S) System Board Fan speed data in fan number
order */
#define WS_SYS_LCD1    0x03030500      /* (S) Value to display on LCD Top line */
#define WS_SYS_LCD2    0x03030600      /* (S) Value to display on LCD Bottom line */
#define WS_SB_LCD_STRING       0x03030700      /* (S) Low Level LCD Display string at current
position */
#define WS_SYS_MESSAGE         0x03030800      /* (S) Value to stored from LCD Messages */
#define WS_NMI_REQ     0x04010100      /* (L) NMI Request bit */
#define WS_SB_CPU_FAULT        0x04010200      /* (L) CPU Fault Summary */
#define WS_SB_FLASH_ENA        0x04010300      /* (L) Indicates FLASH ROW write enabled */
```

-46-

4 7

```
       #define WS_SB_FRU_FAULT        0x04010400      /* (L) Indicates the FRU status */
       #define WS_SB_JTAG     0x04010500      /* (L) Enables JTAG chain on system board */
       #define WS_SYSFAULT    0x04010600      /* (L) System Fault Summary */
       #define WS_SYS_OVERTEMP        0x04010700      /* (L) Indicates Overtemp fault */
 5     #define WS_CAN1_FAN_SYSFLT             0x04010800       /* (L) Indicates Canister #1 Fan System Fault
       */

       #define WS_CAN2_FAN_SYSFLT             0x04010900       /* (L) Indicates Canister #2 Fan System Fault
       */

       #define WS_CAN3_FAN_SYSFLT             0x04010A00       /* (L) Indicates Canister #3 Fan System Fault
10     */

       #define WS_CAN4_FAN_SYSFLT             0x04010B00       /* (L) Indicates Canister #4 Fan System Fault
       */
       #define WS_NMI_MASK            0x04020100      /* (B) CPU NMI processor mask (LSB=CPU1) */
       #define WS_SB_CPU_ERR         0x04020200      /* (B) CPU Error bits (LSB = CPU1) */
15     #define WS_SB_CPU_POK         0x04020300      /* (B) CPU Power OK (LSB = CPU1) */
       #define WS_SB_CPU_PRES        0x04020400      /* (B) CPU Presence bits (LSB = CPU1) */
       #define WS_SB_CPU_TEMP        0x04020500      /* (B) CPU Thermal fault bits (LSB = CPU1) */
       #define WS_SI_EVENTS   0x10050100      /* (E) System Interface Event Queue */
       #define WS_RI_CD        0x11010100      /* (L) Status of Remote Port Modem CD */
20     #define WS_RI_CTS       0x11010200      /* (L) Status of Remote Port Modem CTS */
       #define WS_RI_DSR       0x11010300      /* (L) Status of Remote Port Modem DSR */
       #define WS_RI_DTR       0x11010400      /* (L) State of Remote Port Modem DTR */
       #define WS_RI_RTS       0x11010500      /* (L) Status of Remote Port Modem RTS */
       #define WS_RI_CALLOUT          0x11020100      /* (B) Controls Call out Script activation */
25     #define WS_CALLOUT_STATUS            0x11020200       /* (B) Read Call Out Status */
       #define WS_RI_EVENTS          0x11050100      /* (E) Remote Interface Event Queue */
       #define WS_CAN_FAN_HI         0x20010100      /* (L) Canister Fans HI */
       #define WS_CAN_FAN_LED       0x20010200      /* (L) Canister Fan Fault LED */
       #define WS_CAN_POWER         0x20010500      /* (L) Controls canister PCI slot power */
30     #define WS_CAN_S5_PRESENT    0x20010600      /* (L) Indicates the presence of something in slot 5
       */
       #define WS_SYS_CAN_TYPE      0x20010700      /* (L) Set System Type (0: Raptor16 or 1:Raptor 8)
       */
       #define WS_CAN_FAN_LOLIM     0x20020100      /* (B) Fan low speed fault limit */
35     #define WS_CAN_PCI_PRESENT           0x20020200       /* (B) Reflects PCI card slot[1..4] presence
       indicator pins ( MSB to LSB) 4B,4A,3B,3A,2B,2A,1B,1A */
       #define WS_CAN_FANFAULT      0x20020300      /* (B) Canister Fan Fault Bits */
       #define WS_PCI_SLOT_PWR      0x20020400      /* (B) Turn on/off PCI Slot of Raptor 8 */
       #define WS_CAN_FAN_DATA      0x20030300      /* (S) Canister Fan speed data */
40

       /* ************************************************************ */
       /* This is the Wire Service Attributes for named items. */
       /* The attribute information is stored in a symbolic constant named the same */
45     /* as the named item then followed by two underscores */
       /* */
       /* Attributes consist of: */
       /*      R/W access for internal WS (I), BIOS/OS (O), administrator (A), and general (G) */
       /*      groups. ( 0 = NoAccess 1 = Read Only, 2 = Write Only, 3 = Read/Write ) */
50     /*   */
       /*      maximum possible reques/response length of item in bytes (LL) */
       /* */
       /*      Group Name ID (ID) */
       /* */
55     /*                              IOAGLLID */
```

```c
#define WS_DESCRIPTION__        0x11114000    /* (S) Wire Service Processor Type/Description */
#define WS_REVISION__           0x11112000    /* (S) Wire Service Software Revision/Date Info */
#define WS_WDOG_CALLOUT__                 0x33310100    /* (L) This is a bit controlling callout on a
watchdog timeout. */
#define WS_WDOG_RESET__         0x33310100    /* (L) This is a bit controlling system on a watchdog
timeout. */
#define WS_NVRAM_RESET__        0x22200100    /* (B) Trigger to reset NVRAM Data */
#define WS_SYS_BOOTFLAG1__                0x33310100    /* (B) System Boot Flag 1 */
#define WS_SYS_BOOTFLAG2__                0x33310100    /* (B) System Boot Flag 2 */
#define WS_SYS_BOOTFLAG3__                0x33310100    /* (B) System Boot Flag 3 */
#define WS_SYS_BOOTFLAG4__                0x33310100    /* (B) System Boot Flag 4 */
#define WS_SYS_XDATA_KBYTES__             0x11110100    /* (B) Size of the WS_SYS_XDATA in
kilobytes */
#define WS_SYS_XDATA__          0x3331ff00    /* Byte Array for storage of arbitrary external data
in NVRAM */
#define WS_NVRAM_FAULTS__                 0x11110100       /* (B) Faults detected in NVRAM Data */
#define WS_SYS_LOG__            0x3311ff00    /* System Log */
#define WS_RI_QUEUE__           0x3300ff00    /* (Q) Queue of data going to Remote Interface */
#define WS_SI_QUEUE__           0x3300ff00    /* (Q) Queue of data going to System Interface */
#define WS_SYS_SCREEN__         0x3311ff00    /* System Screen */
#define WS_CALLOUT_SCRIPT__               0x3330ff00       /* (S) The callout script for remote
notification */
#define WS_PASSWORD__           0x33301000    /* (S) The access password for Wire Service */
#define WS_SYS_BP_SERIAL__      0x31111000    /* (S) Last known Back Plane serial data */
#define WS_SYS_CAN_SERIAL1__              0x31111000    /* (S) Last known Canister 1 Serial data */
#define WS_SYS_CAN_SERIAL2__              0x31111000    /* (S) Last known Canister 2 Serial data */
#define WS_SYS_CAN_SERIAL3__              0x31111000    /* (S) Last known Canister 3 Serial data */
#define WS_SYS_CAN_SERIAL4__              0x31111000    /* (S) Last known Canister 4 Serial data */
#define WS_SYS_RI_SERIAL__      0x31111000    /* (S) Last known Remote Interface serial data */
#define WS_SYS_SB_SERIAL__      0x31111000    /* (S) Last known System Board serial data */
#define WS_SYS_PS_SERIAL1__               0x31111000    /* (S) Last known Power Supply 1 serial data
*/
#define WS_SYS_PS_SERIAL2__               0x31111000    /* (S) Last known Power Supply 2 serial data
*/
#define WS_SYS_PS_SERIAL3__               0x31111000    /* (S) Last known Power Supply 3 serial data
*/
#define WS_NAME__        0x33312000    /* (S) System Identifying Name */
#define WS_BOOTDEVS__           0x3331ff00    /* (S) BIOS Boot drive information */
#define WS_SYS_LOG_CLOCK__                0x11110400    /* (S) Current time from log timestamp clock
(seconds) */
#define WS_SYS_LOG_COUNT__                0x11110200    /* (S) Number of Log entries */
#define WS_MODEM_INIT__         0x33315000    /* (S) Modem initialization string */
#define WS_EVENT_ID01__         0x31111000    /* (S) Canister Change Event */
#define WS_EVENT_ID02__         0x31111000    /* (S) Power Supply Change Event */
#define WS_EVENT_ID03__         0x31111000    /* (S) Queue Event */
#define WS_EVENT_ID04__         0x31111000    /* (S) Temp Warn or Shut Event */
#define WS_EVENT_ID05__         0x31111000    /* (S) ACOK Change Event */
#define WS_EVENT_ID06__         0x31111000    /* (S) DCOK Change Event */
#define WS_EVENT_ID07__         0x31111000    /* (S) Fan Fault Event */
#define WS_EVENT_ID08__         0x31111000    /* (S) Screen Event */
#define WS_EVENT_ID09__         0x31111000    /* (S) CPU Fault Event */
#define WS_EVENT_ID0A__         0x31111000    /* (S) OS_TimeOut Event */
#define WS_CALLOUT_MASK__                 0x31110200    /* (S) Call Out Masking string */
#define WS_BIOS_REV__           0x31111000    /* (S) Storage of current BIOS Revision */
```

49

```c
#define WS_SYS_POWER__          0x33310100      /* (L) Controls system master power S4_POWER_ON
*/
#define WS_SYS_REQ_POWER__         0x22200100      /* (L) Set to request main power on */
#define WS_BP_P12V__   0x11110100      /* (B) Analog Measure of +12 volt main supply */
#define WS_BP_P3V__    0x11110100      /* (B) Analog Measure of +3.3 volt main supply */
#define WS_BP_N12V__   0x11110100      /* (B) Analog Measure of -12 volt main supply */
#define WS_BP_P5V__    0x11110100      /* (B) Analog Measure of +5 volt main supply */
#define WS_BP_VREF__   0x11110100      /* (B) Analog Measure of VREF */
#define WS_SYS_BP_TYPE__    0x11110100      /* (B) Type of system backplane currently only two
types Type 0= 4 canister (small) and Type 1= 8 canister (large) */
#define WS_SYS_CAN_PRES__   0x11110100      /* (B) Presence bits for canisters (LSB=1, MSB=8)
*/
#define WS_SYS_PS_ACOK__    0x11110100      /* (B) Power supply ACOK status (LSB=1, MSB=3)
*/
#define WS_SYS_PS_DCOK__    0x11110100      /* (B) Power supply DCOK status (LSB=1, MSB=3)
*/
#define WS_SYS_PS_PRES__    0x11110100      /* (B) Presence bits for power supplies (LSB=1,
MSB=3) */
#define WS_SYS_RSTIMER__    0x33310100      /* (B) Used to delay reset/run until power stabilized
*/
#define WS_SYS_TEMP_SHUT__     0x33310100      /* (B) Shutdown temperature. Initialized to
??? */
#define WS_SYS_TEMP_WARN__     0x33310100      /* (B) Warning temperature. Initialized to ???
*/
#define WS_SYS_WDOG__       0x33110100      /* (B) System watchdog timer */

/* First issues following command in phase 2 */

#define WS_OS_RESOLUTION_16__      0x33110100      /* (B) Set Resolution (0,1,2,3) of Timer1 */
#define WS_OS_COUNTER_16__         0x33110100      /* (B) Set Counter from (00 - FFh) of Timer1
*/

/* If either operation's failed that it will response error code "02h" back, then try raptor 8 and future
command */

#define WS_OS_RESOLUTION_8__        0x33110100      /* (B) Set Resolution (0,1,2,3) of Timer1 */
#define WS_OS_COUNTER_8__    0x33110100      /* (B) Set Counter from (00 - FFh) of Timer1 */

/* If it's failed it is raptor 16 phase 1 that does not support watchdog */

#define WS_SYS_TEMP_DATA__         0x11110500      /* (S) Temperatures of all sensors on
temperature bus in address order */
#define WS_SB_FAN_HI__      0x33310100      /* (L) System Board Fans HI */
#define WS_SB_FAN_LED__     0x33110100      /* (L) System Board Fan Fault LED */
#define WS_SYS_RUN__        0x33310100      /* (L) Controls the system halt/run line
S1_OK_TO_RUN. */
#define WS_SYS_SB_TYPE__    0x33310100      /* (L) Set System Type (0:Raptor 16 or 1:Raptor 8)
*/
#define WS_SB_BUSCORE__     0x33110100      /* (B) System Board BUS/CORE speed ratio to use
on reset */
#define WS_SB_FANFAULT__    0x33110100      /* (B) System Board Fan fault bits */
#define WS_SB_FAN_LOLIM__   0x33310100      /* (B) Fan speed low speed fault limit */
#define WS_SB_LCD_COMMAND__        0x22000100      /* (B) Low level LCD Controller Command
*/
#define WS_SB_LCD_DATA__    0x22000100      /* (B) Low level LCD Controller Data */
```
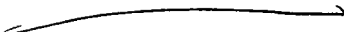
```
        #define WS_LCD_MSG__          0x33110100    /* (B) Send a Byte of Fault Bits from Monitor-B to
        Monitor-A */
        #define WS_SB_DIMM_TYPE__     0x11111000    /* (S) The type of DIMM in each DIMM socket as
        a 16 byte string */
5       #define WS_SB_FAN_DATA__      0x11110600    /* (S) System Board Fan speed data in fan number
        order */
        #define WS_SYS_LCD1__         0x33311000    /* (S) Value to display on LCD Top line */
        #define WS_SYS_LCD2__         0x33311000    /* (S) Value to display on LCD Bottom line */
        #define WS_SB_LCD_STRING__    0x22004000    /* (S) Low Level LCD Display string at current
10      position */
        #define WS_SYS_MESSAGE__      0x11112000    /* (S) Value to stored from LCD Messages */
        #define WS_NMI_REQ__          0x22200100    /* (L) NMI Request bit */
        #define WS_SB_CPU_FAULT__     0x11110100    /* (L) CPU Fault Summary */
        #define WS_SB_FLASH_ENA__     0x33310100    /* (L) Indicates FLASH ROW write enabled */
15      #define WS_SB_FRU_FAULT__     0x33110100    /* (L) Indicates the FRU status */
        #define WS_SB_JTAG__  0x33310100    /* (L) Enables JTAG chain on system board */
        #define WS_SYSFAULT__         0x33110100    /* (L) System Fault Summary */
        #define WS_SYS_OVERTEMP__     0x11110100    /* (L) Indicates Overtemp fault */
        #define WS_CAN1_FAN_SYSFLT__          0x33110100      /* (L) Indicates Canister #1 Fan System Fault
20      */
        #define WS_CAN2_FAN_SYSFLT__          0x33110100      /* (L) Indicates Canister #2 Fan System Fault
        */
        #define WS_CAN3_FAN_SYSFLT__          0x33110100      /* (L) Indicates Canister #3 Fan System Fault
        */
25      #define WS_CAN4_FAN_SYSFLT__          0x33110100      /* (L) Indicates Canister #4 Fan System Fault
        */
        #define WS_NMI_MASK__         0x33310100    /* (B) CPU NMI processor mask (LSB=CPU1) */
        #define WS_SB_CPU_ERR__       0x11110100    /* (B) CPU Error bits (LSB = CPU1) */
        #define WS_SB_CPU_POK__       0x11110100    /* (B) CPU Power OK (LSB = CPU1) */
30      #define WS_SB_CPU_PRES__      0x11110100    /* (B) CPU Presence bits (LSB = CPU1) */
        #define WS_SB_CPU_TEMP__      0x11110100    /* (B) CPU Thermal fault bits (LSB = CPU1) */
        #define WS_SI_EVENTS__        0x33001000    /* (E) System Interface Event Queue */
        #define WS_RI_CD__   0x33110100    /* (L) Status of Remote Port Modem CD */
        #define WS_RI_CTS__  0x33110100    /* (L) Status of Remote Port Modem CTS */
35      #define WS_RI_DSR__  0x33110100    /* (L) Status of Remote Port Modem DSR */
        #define WS_RI_DTR__  0x33110100    /* (L) State of Remote Port Modem DTR */
        #define WS_RI_RTS__  0x33110100    /* (L) Status of Remote Port Modem RTS */
        #define WS_RI_CALLOUT__       0x33310100    /* (B) Controls Call out Script activation */
        #define WS_CALLOUT_STATUS__          0x33310100      /* (B) Read Call Out Status */
40      #define WS_RI_EVENTS__        0x33002000    /* (E) Remote Interface Event Queue */
        #define WS_CAN_FAN_HI__       0x33310100    /* (L) Canister Fans HI */
        #define WS_CAN_FAN_LED__      0x33310100    /* (L) Canister Fan Fault LED */
        #define WS_CAN_POWER__        0x33310100    /* (L) Controls canister PCI slot power */
        #define WS_CAN_S5_PRESENT__           0x11110100      /* (L) Indicates the presence of something
45      in slot 5 */
        #define WS_SYS_CAN_TYPE__     0x33310100    /* (L) Set System Type (0: Raptor 16 or 1: Raptor
        8) */
        #define WS_CAN_FAN_LOLIM__            0x33310100      /* (B) Fan low speed fault limit */
        #define WS_CAN_PCI_PRESENT__          0x11110100      /* (B) Reflects PCI card slot[1..4] presence
50      indicator pins ( MSB to LSB) 4B,4A,3B,3A,2B,2A,1B,1A  */
        #define WS_CAN_FANFAULT__     0x11110100    /* (B) Canister Fan Fault Bits */
        #define WS_PCI_SLOT_PWR__     0x33310100    /* (B) Turn on/off PCI Slot of Raptor 8 */
        #define WS_CAN_FAN_DATA__     0x11110200    /* (S) Canister Fan speed data */
        #ifndef FAR_POINTERS
55      #ifndef NEAR_POINTERS
```

51

```
#include "***ERROR - Pointer Type not defined"
#endif
#endif
```

52